VŠB – Technická univerzita Ostrava
Fakulta elektrotechniky a informatiky
Katedra informatiky

Modul ověřující jedinečnost projektů

Module for Project's Uniqueness Checking

2009

Radek Stromský

Prohlašuji, že jsem tuto diplomovou práci vypracoval samostatně. Uvedl jsem všechny literární prameny a publikace, ze kterých jsem čerpal.

V Ostravě 4. května 2009 .....

Rád bych na tomto místě poděkoval všem, kteří mi s prací pomohli, především pak týmu lidí, pracujících na vývoji systému Maus, za jejich odhodlání a spolupráci.

Abstrakt

Tato diplomová práce se zabývá problémem plagiátorství zdrojových kódů u studentských projektů. Hlavním cílem je analyzovat problémy při hledání plagiátů a dále pak navrhnout a implementovat takové řešení, které by napomohlo odhalit plagiáty mezi skupinou vybraných projektů. Výsledným produktem praktické části bude modul pro ověřování jedinečnosti projektů v programovacím jazyce C#. Tento modul bude univerzitnímu systému Maus poskytovat svou funkcionalitu prostřednictvím webových služeb.

Pro vývoj modulu byl vybrán .NET framework 3.5 a jazyk C#. Společné datové úložiště pro systém Maus poskytuje MS SQL Server 2008. Pro analyzování aplikací ve formátu ECMA CIL byla vybrána knihovna *Cecil* z projektu *Mono*.

Klíčová slova: plagiát, plagiátorství, detekce, porovnání, podobnost, zdrojový kód, .NET, C#

Abstract

This diploma thesis aims to the sphere of source code plagiarism in the frame of student projects. The main objective is to analyze all the problems related to the plagiarism detection and to design and implement an appropriate solution providing support for the code plagiarism revealing among a specific group of student projects. This thesis final product is going to be delivered in the form of specialized C# module for the project uniqueness verification. Such a module will provide its functionality to the Maus system using the web services interface.

.NET Framework 3.5 and C# language have been chosen for development purpose of the modul. MS SQL Server 2008 gives suport of common data storage to the Maus system. *Cecil* library(part of the *Mono* project) has been chosen for analyzing applications in ECMA CIL format.

Keywords: plagiarism, detection, comparison, similarity, source code, .NET, C#

Seznam použitých zkratek a symbolů

ASM	–	Approximate String Matching
CIL	–	Common Intermediate Language
CLI	–	Common Language Infrastructure
ECMA	–	European Computer Manufacturers Association
GST	–	Greedy String-Tiling algorithm
IL	–	Intermediate Language
MSIL	–	Microsoft Intermediate Language
RKR-GST	–	Running Karp-Rabin Greedy String-Tiling algorithm
RTTI	–	Run-Time Type Information

Obsah

Seznam tabulek

Seznam obrázků

Seznam výpisů zdrojového kódu

1 Úvod

Jedním z nejdůležitějších cílů při studiu informatiky, je pochopení programovacích technik. Znalost těchto technik je v převážné většině zjišťována zadáváním samostatných studentských projektů, na kterých si studenti zároveň osvojí vyučovanou látku. Při následném hodnocení těchto projektů se může nabídnout otázka, zda student opravdu pracoval na projektu samostatně. Odpovědět na tuto otázku nebude rozhodně jednoduché. Pedagog při hodnocení projektů, podrobně prochází zdrojové kódy, ale podobnost mezi jednotlivými projekty mu může uniknout.

Tato diplomová práce si klade za cíl zjednodušit vyučujícím kontrolu studentských projektů takovým způsobem, že toto porovnávání zdrojových kódů a hledání případných plagiátů na určité úrovni automatizuje.

Druhá kapitola je úvodem do problému plagiátorství a zmiňuje některé existující systémy. Třetí kapitola se zabývá analýzou možných řešení a z těch stanovuje to řešení, které bude cílem praktické části této práce. Čtvrtá kapitola se věnuje návrhu mudulu, který bude implementovat vybrané řešení. Pátá kapitola se zabývá testováním modulu a vytvářením konfigurací.

2 Plagiátorství zdrojových kódů

Za plagiát se dá označit takový zdrojový kód, který byl vytvořen zkopírováním jiného zdrojového kódu, jeho částí, nebo jeho částečnou úpravou a to bez vědomí autora, či uvedení reference na původní zdroj. U většiny případů je plagiát vytvořen bez větších známek pochopení zdrojového kódu a od toho pak plynou druhy úprav, které jsou plagiátorem nejčastěji prováděny. Faidhi a Robinson definují šest úrovní úprav programu [3]. Tyto úrovně jsou shrnuty v tabulce 1.

Úroveň	**Druh úpravy**
0	originální program
1	komentáře a odsazení
2	názvy identifikátorů
3	umístění proměnných
4	kombinace funkcí
5	příkazy programu
6	řídící logika programu

Tabulka 1: Druhy úprav při tvorbě plagiátů

Význam prvních čtyř úrovní je víceméně jednoznačný. Úroveň 4 chápejme jako rozdílné pořadí funkcí, ale také jako zanořování funkcí, nebo jejich spojování. Úrovní 5 jsou myšleny záměny příkazů za jejich ekvivalenty(bráno po funkční stránce). Do úrově 6 patří úpravy řídící logiky programu, které jinou cestou dosáhnou stejného výsledku.

Při detekci plagiátů je težké odhalit upravené zdrojové kódy, které spadají do úrovně 5. a 6. Jedním důvodem je, že nelze jednoduše zjistit, zdali jeden projekt opravdu vznikl modifikací druhého, nebo se jedná o originální řešení se stejnou myšlenkou. Jelikož je tato práce zaměřena na plagiátorství u studentských projektů, musíme také brát v úvahu, že již zadání určuje směr, jimž se má student udávat. Z tohoto důvodu by většina projektů bezesporu automaticky spadala do šesté úrovně, popřípadě úrovně páté a to i za takových okolností, kdyby se o plagiát nejednalo. V této práci se proto budu nadále snažit dosáhnout pokrytí úprav do úrovně 4 včetně.

Úpravy čtvrté úrovně se dají označit za klíčové. Úpravy spadající do nižších úrovní není příliš těžké odhalit a naopak, odhalit úpravy vyšších úrovní je téměř nemožné. Ukázky zdrojových kódů ve výpisech 1 a 2 dávají prvotní pohled na druh úprav, které se pod

úrovní 4 skrývají.

```
public class MyMath
{
    public void Square(double number)
    {
        double result = System.Math.Pow(number, 2);
        System.Console.WriteLine("Výsledek: {0}", result);
    }
}
```

Výpis 1: Přepis spojováním

```
public class MyMath
{
    public void Square(double number)
    {
        double result = System.Math.Pow(number, 2);
        Display(result.ToString());
    }
    public void Display(string text)
    {
        System.Console.WriteLine("Výsledek: {0}", text);
    }
}
```

Výpis 2: Přepis zanořením

Pokud se detailně podíváme na tyto zdrojové kódy, můžeme usoudit, že metoda `Square` ve výpisu 1 mohla vzniknout kombinací dvou metod uvedených ve výpisu 2 a to dosazením kódu metody `Display` na místo jejího volání (řádek 6, výpis 2). V úvahu přichází i opačný způsob úpravy, kdyby kód výpisu 2 byl vytvořen rozložením metody `Square` z výpisu 1. V obou případech však můžeme říct, že se jedná o velice podobné řešení.

Pro ochranu autorských práv bylo v minulosti vyvinuto mnoho rozličných systémů. První z nich byly zaměřeny na detekci plagiátorství u textových dokumentů. Na speciální formu autorského díla, kterou zdrojový kód je, však jednoduše nelze použít textově založené detekční techniky. Ty totiž neberou v úvahu možné úpravy specifické právě pro zdrojové kódy(viz tabulka 1). Prechelt a kolektiv zavedli dvě hlavní kategorie technik pro detekci plagiátorství u zdrojových kódů [1].

Porovnávání rysů (*Feature comparison*)
Systémy založené na tomto řešení počítají pro každý program hned několik rozdílných metrik. Každý program je pak reprezentován bodem, který se nachází v n-rozměrném kartézkém prostoru, kde n je počet metrik. Programy, které pak leží v prostoru blízko sebe, jsou brány za potenciální plagiáty. První takový systém uvedl Karl J. Ottenstein[4] v roce 1976. Jeho systém využíval čtyři základní Halstead metriky [2]. Pozdější systémy pak uvedly mnohem větší množství metrik, výsledky i tak nebyly stále uspokojivé. Při procesu extrahování metrik bylo opomíjeno mnoho důležitých informací, které vypovídaly o struktuře programu. Tento nedostatek by se nedal odstranit ani přidáním dalších metrik. Systém by byl buď příliš citlivý a tím by vznikala spousta falešných nálezů plagiarismu, nebo by naopak i jednoduchá úprava programu dokázala systém oklamat.

Porovnávání struktury (*Structure comparison*)
Jedná se o řešení, porovnávající samotnou strukturu programu a nejen součty hodnot popisující jednotlivé aspekty kódu. Proces porovnávání převážně probíhá tak, že se program převede na proud tokenů, který se následně využívá k porovnání a vyhledávání shodných úseků.

Některé systémy vznikly kombinací těchto dvou přístupů a jsou označovány za tzv. *hybridní řešení*.

2.1 Přehled existujících řešení

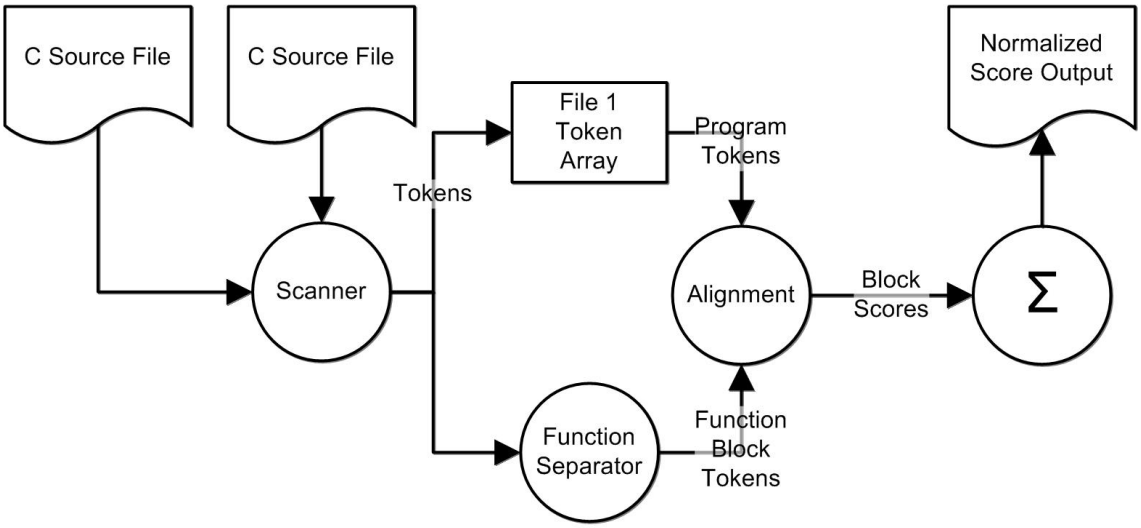
Během studia literatury zaobírající se problémem plagiátorství jsem narazil na několik existujících řešení. Tři z nich jsou krátce popsány v následujících kapitolách.

2.1.1 SIM

Pro hledání podobností mezi programy, systém *SIM* využívá vyhodnocování jejich správnosti, stylu a jedinečnosti. Každý program je nejprve analyzován lexikálním analyzátorem, který vytváří proud tokenů. Ty jsou reprezentovány celočíselnou hodnotou. Tokeny pro klíčová slova, speciální znaky a komentáře jsou předem určeny, zatímco tokeny pro identifikátory jsou přiřazeny dynamicky a uloženy ve sdílené tabulce tokenů. Proud tokenů z druhého programu je pak rozdělen do *balíčků*, které reprezentují tokeny jednotlivých funkcí programu. Každý balíček je samostatně přiřazen k řetězci tokenů prvního programu a za pomoci vkládání mezer se pak upraví řetězec tak, aby oba

měly stejnou délku. Tímto postupem je však možné dosáhnout mnoha rozdílných *zarovnání*. Algoritmus *zarovnání řetězce* používá bodovací systém, který pevně stanoveným způsobem určuje, které z možných zarovnání odpovídá prvnímu programu nejvíce [5, 6].

Toto řešení zohledňuje úpravy do úrovně 4 (viz tabulka č.1). V rámci čtvrté úrovně nebere v úvahu možné spojování metod, nebo jejich zanořování.



Obrázek 1: Princip systému SIM

2.1.2 YAP3

Michael J. Wise uvedl v roce 1996 další systém na detekci plagiátorství založený na porovnávání struktury programů. Porovnávání u systému YAP3 je rozloženo do dvou hlavních fází. V první fázi se z obou porovnávaných programů vygeneruje posloupnost tokenů. V průběhu generování se odfiltrují komentáře a řetězce(konstanty typu *string*) a všechny znaky se převedou na malá písmena. V druhé fázi se za pomoci RKR-GST algoritmu [1, 7] vyhledávají nejdelší a zároveň nepřesahující shodné pasáže [5, 7].

YAP3 dokáže odhalit úpravy až 4.úrovně (viz tabulka č.1), včetně možných volání zanořených metod.

2.1.3 JPlag

JPlag je detekční systém s webovým rozhraním, který vyhledává dvojice podobných programů z určité množiny programů. Je založen na podobném algoritmu jako YAP3. Zdrojové kódy jsou nejprve zpracovány lexikálním analyzátorem, tudíž se opět využívá proudu tokenů. Pro zamezení náhodné podobnosti systém definuje rozdílné tokeny pro složené závorky (“{” a “}”) na základě jejich významu, aby se jednoduše dalo rozpoznat zda se jedná o závorky ohraničující funkci či nikoliv. Mezery, komentáře a názvy identifikátorů jsou ignorovány. Na takto upravené řetězce tokenů je pak aplikován GST algoritmus, který nalezne nejdelší, nepřesahující sekvenci stejných tokenů. Výsledek porovnávání je posléze prezentován uživateli ve formě HTML stránky [1, 5].

JPlag v současnosti podporuje programovací jazyky Java, C, C++, Scheme a je dostupný na webové adrese `https://www.ipd.uni-karlsruhe.de/jplag/`.

3 Analýza problému

Systémy popsané v předcházející kapitole byly testovány a využívány v praxi pro detekci plagiátorství zdrojových kódů. Veškeré tyto systémy zpracovávají textovou formu programu (tj. zdrojové kódy), popřípadě jejich předzpracovanou formu.

Jedním z bodů této práce je právě navrhnout a implementovat nové řešení takového systému. V práci se zaměřuji na využití nového přístupu pro zkoumání aplikací a tím je proces reflexe. Tudíž bude pro porovnávání a hledání podobností využíváno programů přeložených do jazyka CIL.

3.1 Common Intermediate Language

Tento *společný mezijazyk* patří do nejnižší úrovně pro člověka srozumitelných jazyků. Všechny jazyky platformy .NET jsou kompilovány právě do CIL. Jedná se o objektově orientovaný *assembly* jazyk spouštěný virtuálním strojem. CIL je součástí standardu ECMA-335, který popisuje celou infrastrukturu CLI[8]. Jazyk CIL byl původně uveden s příchodem beta verze jazyků pro .NET a to pod názvem *Microsoft Intermediate Language(MSIL)*[9, 10].

Pro potřebu této práce však není zapotřebí jazyk CIL příliš znát. Povědomí o jeho existenci bude stačit a pro získávání informací o programu z jeho zkompilované formy se využije procesu reflexe.

Reflexe je mechanismus, který umožňuje za běhu zkoumat, upravovat, nebo využívat jednotlivých součástí programu. Reflexe je plně podporována u platforem využívajících IL (Intermediate Language) jako jsou *.NET*, nebo *Java*. Například jazyk C++ reflexy dlouhou dobu nepodporoval. Následně byla reflexe implementována i pro C++ a to pod akronymem *RTTI (Run-Time Type Information)*.

3.2 Zavedení pojmů

Než se začnu věnovat samotnému výkladu, je nutné zavést si několik pojmů, aby nedošlo k záměně jejich významu.

Projekt - studentský projekt, který je reprezentován jedním či několika C# projekty (.csproj) a s nimi souvisejícími zdrojovými kódy(.cs), popřípadě jejich zkompilovanou formou (.dll a .exe).

Assembly - zkompilovaná forma projektu, nebo jeho část (.dll a .exe).

Aplikace/Program - spustitelná verze projektu, obsahující jednu nebo více assembly, popřípadě další soubory, nutné pro správný běh aplikace.

Zdrojový kód - nezkompilovaná forma projektu (.csproj, .cs).

Forma - formát projektu. Ten může být buď zkompilovaný (pak se jedná o tzv. Assembly), nebo nezkompilovaný (viz. pojem *„Zdrojový kód“*).

Prvek - objekt popisující část zdrojového kódu, jako jsou například: třída, metoda, parametr metody atd.

3.3 Hledání podobnosti

Hledání podobnosti lze popsat jako proces, na jehož vstupu se nachází dva projekty v některé z jejich forem a na jeho výstupu hodnota popisující procentuální shodu mezi těmito dvěma projekty. Způsobů, jakými lze z těchto vstupů získat relevantní výstup, může být hned několik.

Pokud jde o metodiky detekce plagiátorství, nabízí se jejich rozdělení dle následujících kritérií:

- dle formy projektů vstupujících do procesu porovnávání (dále jen *kritérium vstupních forem*)
- dle způsobu zpracování porovnávaných informací (dále jen *kritérium porovnávání*)

Za pomocí *kritéria vstupních forem* lze definovat právě dva množné druhy řešení:

1. **Zpracování samotných *zdrojových kódů***

   Právě toto řešení je implementováno systémy SIM[6], YAP3[7] a JPlag[1], které jsou uvedeny v kapitole 2.1. Tento přístup má výhodu v kompletnosti vstupních informací - do procesu porovnávání vstupuje přesně to, co autor naprogramoval. Prvním krokem těchto řešení je v převážné většině převedení zdrojového kódu na proud, nebo sekvenci tokenů. Během této konverze jsou často odfiltrovány nežádoucí tokeny, jako jsou například komentáře, mezery, textové konstanty a někdy i názvy identifikátorů či metod.

2. **Získávání informací z *assembly***

   K tomu se využívá tzv. reflexe. Jelikož se v tomto případě bude porovnávat forma projektu vytvořena kompilátorem, může dojít, a také ve většině případů dochází, k ovlivnění výsledku porovnávání. Tato odchylka hodnoty výsledku nemusí být nutně negativního charakteru. Pokud si uvědomíme, že na informacích, o které přicházíme při kompilaci příliš nezáleží, bude nám jasné, že výstupem této metody budou stále věrohodná data. Tento přístup tak velice efektivně nahrazuje proces "*tokenizace*", jenž je za jiných okolností nutný při přímém porovnávání zdrojových kódů.

Kritériem porovnání lze definovat několik základních přístupů k procesu porovnávání. Na nejvyšší úrovni lze přístupy rozdělit, jak již bylo zmíněno dříve, na *Porovnávání rysů* a *Porovnávání struktury*. Jelikož se metoda porovnávání rysů jeví jako nedostatečné řešení, v této práci se již budu zabývat výhradně technikami založenými na metodě porovnávání struktury programu.

Níže zmíněné přístupy vyplývají z použitých technik existujících řešení(viz kapitola 2), ale také z teoretických úvah vycházejících z mých osobních zkušeností.

1. **Lineární analýza kódu**

   Jedná se o řešení aplikované v systémech SIM[6], YAP3[7] a JPlag[1]. Původně tato řešení spadala do skupiny metod *porovnávání struktur*(viz kapitola 2)[1]. Jelikož jsou však tyto systémy založeny na porovnávání dvou lineárních sekvencí tokenů, bylo by používání výrazu "*strukurální analýza*" příliš zavádějící. *Lineární analýza kódu* představuje aplikaci porovnávácího algoritmu na výstup z lexikálního analyzátoru. Lexikální analyzátor, neboli skener, může využívat speciálních tokenů, které mohou později sloužit k rychlému nalezení části kódu, nebo jako je tomu v systému JPlag, pro rozdělení proudu tokenů na jednotlivé funkce.

2. **Statická analýza členů**

   Tato technika, jako první zmíněná, je plně založena na porovnávání struktury aplikací. O tom vypovídá fakt, že v první fázi prochází zdrojový kód procesem lexikální, ale i syntaktické a sémantické analýzy. *Statická analýza členů* je založena na detailním zkoumání třídních členů, tj. metod, vlastností, ale i proměnných. Předmětem porovnávání jsou tedy především rysy jednotlivých členů. Těmi mohou být například: jméno metody, vstupní parametry, návratová hodnota, ale také

samotné tělo metody, ve kterém se mohou zkoumat proměnné, primitivní příkazy, nebo třeba volání jiných metod.

Pozn.: Metodika statické analýzy členů byla vybrána za hlavní téma této práce a praktická část je její implementací.

3. **Kompilace statického grafu aplikace**

   Metodika postavená na analýze sémantické struktury aplikace. V první fázi se nalezne vstupní bod aplikace. V četných případech se jedná o statickou metodu `main`. Rekurzivní analýzou metod aplikace, počínající právě ve vstupním bodě, se tvoří orientovaný graf popisující průchod aplikací. Vrcholy grafu reprezentují jednotlivé metody a hrany představují jejich volání. *Statický graf* proto, že nebereme v úvahu možné hodnoty proměnných, které by mohly ovlivnit průchod aplikací při jejím spuštění. Graf tudíž obsahuje veškerá možná volání v rámci aplikace. Pro každou aplikaci se tedy může předem vytvořit graf a skupinu takových grafů pak mezi sebou porovnávat. Hlavním problémem k řešení je právě implementace algoritmu přibližného porovnání dvou orientovaných grafů.

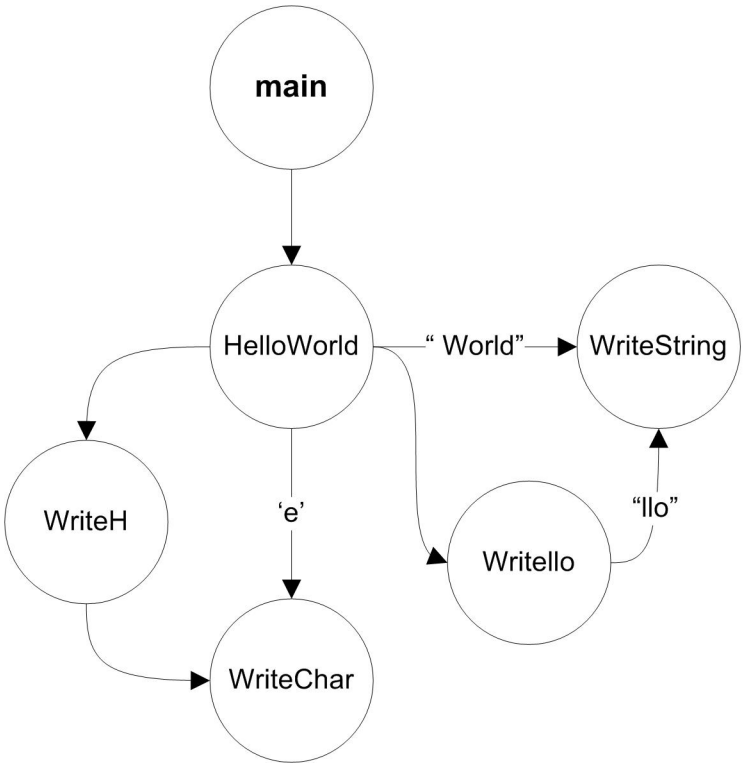
4. **Kompilace dynamického grafu aplikace**

   Myšlenka této metodiky je stejná jako u *Kompilace statického grafu aplikace*, s tím rozdílem, že graf je tvořen při *reálném* průchodu aplikací. Na začátku se definuje množina vstupních hodnot, které se použijí jako vstupní parametry pro spuštění aplikace. Graf pak reprezentuje konkrétní průchod aplikací. Pro jednu aplikaci tak lze vytvořit sadu grafů a ty budou popisovat chování pro různé vstupní hodnoty. Aplikování této metodologie vyžaduje shodné rozhraní u porovnávaných aplikací.

```
class HelloWorldGraph
{
    static  void Main(string[] args){
        HelloWorldGraph hello = new HelloWorldGraph();
        hello.HelloWorld();
    }
    private void HelloWorld(){
        WriteH();
        WriteChar('e');
        Writello ();
        WriteString("␣World");
    }
    private void WriteH(){
        WriteChar('H');
    }
```

```
    private void Writello (){
        WriteString(" llo ");
    }
    private void WriteString(string s){
        Console.Write(s);
    }
    private void WriteChar(char c){
        Console.Write(c);
    }
}
```

Výpis 3: Kompilace statického grafu aplikace



Obrázek 2: Statický graf aplikace z výpisu 3

Výběr metodologie *statické analýzy členů* pro praktickou část této práce má následující opodstatnění:

- I když je *lineární analýza kódu* nejrozšířenější technikou v současně používaných systémech pro detekci plagiátorství, je v této oblasti stále mnohé co vylepšovat. Právě proto jsem se rozhodl vyhnout se zaběhnutým postupů.

- Teorie *kompilace statického grafu aplikace* se zdá být efektivně využitelná pouze u aplikací většího rozsahu. Jelikož se však u studentských projektů jen zřídkakdy objeví rozsáhlá aplikace, byla by tato metodika u většiny případů nevyužitelná. To stejné platí i pro *kompilaci dynamického grafu aplikace*.

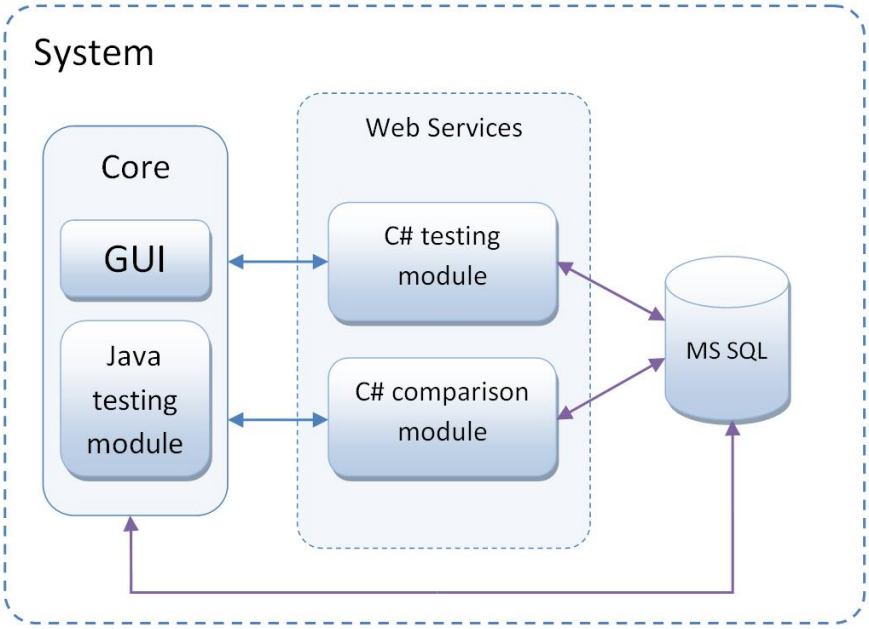
4 Návrh modulu

Cílem praktické části této práce je vytvořit *modul* pro rozpoznávání podobných projektů a poskytnout rozhraní pro podporu komunikace se zbytkem systému *Maus*(viz kapitola 4.1). Modul bude implementován nad platformou .NET a jazyku C# jako samostatná knihovna.

Při návrhu modulu je dále nutné brát v úvahu následující:

1. Modul bude využíván částí systému Maus, postavené na platformě *java*, proto je nutné zavést rozhraní, které bude na platformě nezávislé.
2. Je zapotřebí zavést rozsáhlou množinu konfigurovatelných hodnot, které ovlivní proces porovnávání a tudíž i konečný výsledek.
3. Na základě analýzy prvotních výsledků se definuje několik konfigurací, kde každá může být úpřednostněna na základě porovnávaných projektů.

4.1 Architektura systému



Obrázek 3: Architektura systému

Modul pro porovnávání projektů v jazyce C#, který je tématem této práce, je jedním z modulů systému Maus. Jeho architektura je zachycena na obrázku 3. Jádro systému, jehož autorem je Bc. Radim Velčovský, obsahuje především modul uživatelského rozhraní a modul pro testování projektů v jazyce Java. Autorem modul pro testování projektů v jazyce C# je Bc. Milada Píšová. Funkcionalita modulu pro porovnávání projektů v jazyce C# jakožto i modulu pro testování je poskytnuta systému prostřednictvím webových služeb.

4.2 Návrh komunikace se systémem

Jak již bylo zmíněno dříve, komunikace systému s modulem probíhá prostřednictvím webové služby. Tato kapitola detailně popisuje veškeré webové metody, které jsou službou poskytnuty.

`void CompareProjectsByQuestions(string configName)`

Spustí porovnávání všech dostupných projektů. Porovnávání probíhá po skupinách, které tvoří projekty se stejným zadáním.

`void CompareProjectsByQuestionIds(long[] questionIds, string configName)`

Spustí porovnávání projektů dostupných pro vybraná zadání.

`List<Comparison> GetAllComparisons()`

Vrátí seznam všech porovnání, které byli kdy provedeny.

`List<Comparison> GetComparisons(int skip, int take)`

Vrátí seznam vybraných porovnání. Umožňuje stránkování výsledků.

`ProjectResult GetComparisonDetails(long comparisonId)`

Vrátí detaily pro konkrétní porovnání. Metoda se stará o dekomprimaci a deserializaci dat.

`List<ComparisonConfig> GetAllConfigs()`

Vrátí seznam všech dostupných konfigurací.

`List<FeatureConfig> GetFeatureConfigsById(long id)`

Vrátí seznam nastavení rysů pro konfiguraci určenou identifikačním číslem.

`List<FeatureConfig> GetFeatureConfigsByName(string name)`

Vrátí seznam nastavení rysů pro konfiguraci určenou jejím názvem.

`void RestoreDefaultConfig()`

Vytvoří konfiguraci se jménem "Default" a nastaví všechny váhy rysů na hodnotu 1. Pokud konfigurace se stejným jménem existuje, bude smazána i se všemi souvisejícími daty.

`void CreateNewConfig(string configName)`

Vytvoří novou konfiguraci a pro všechny rysy nastaví váhu na hodnotu 1.

`void SaveFeatureConfig(FeatureConfig config)`

Uloží nastavení váhy pro daný rys aplikace.

`void SaveFeatureException(FeatureException featureException)`

Uloží změny ve výjimce pro daný rys aplikace. Pokud je hodnota ID menší nebo rovna nule, bude pro rys vytvořena výjimka nová.

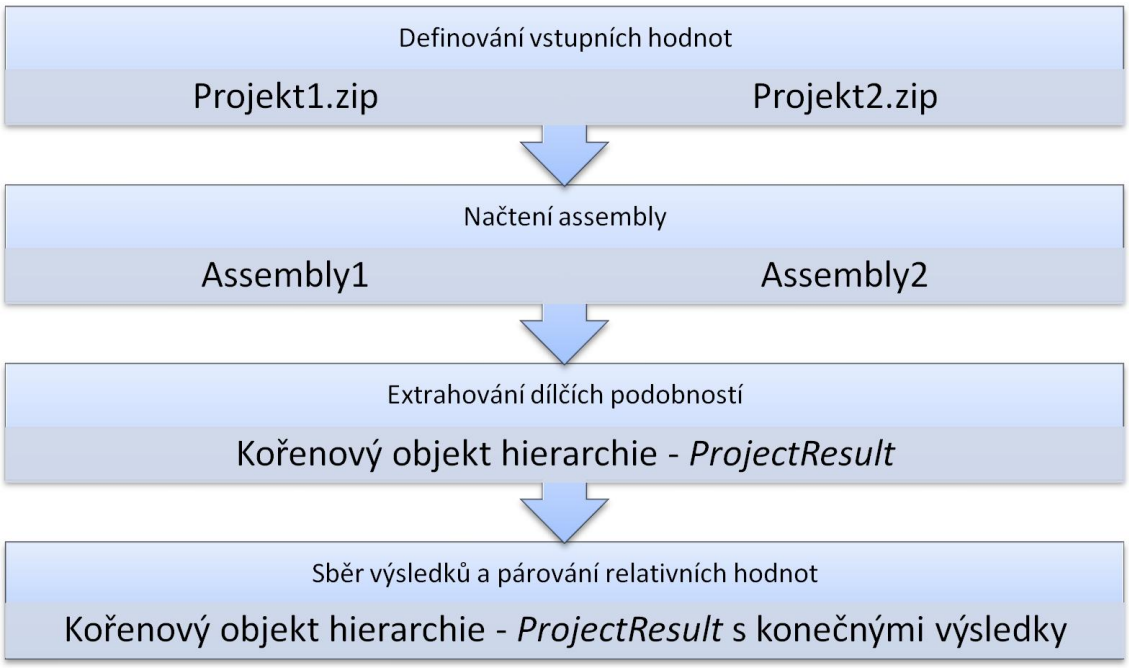
`long GetQuestionIdByComparisonId(long comparisonId)`

Vrátí identifikační číslo otázky(zadání) na základě identifikačního čísla porovnání.

4.3 Proces porovnávání

Průběh porovnávání *statickou analýzou členů* byl již nastíněn v kapitole 3.3. Nyní si tento proces představíme podrobněji.

Jako u jakéhokoliv jiného porovnávání jsou vstupem do procesu právě dvě aplikace(v tomto případě .NET aplikace). Jelikož každý studentský projekt může obsahovat více než jednu assembly, systém Maus poskytuje balík v `zip` formátu, který obsahuje veškeré nutné soubory pro běh aplikace. Výstupem procesu porovnávání je především celé číslo, představující procentuální podobnost dvou projeků. To ovšem není jediná informace vystupující z procesu. Výstupem je celá hierarchie tříd, popisující dílčí podobnosti v procentech pro jednotlivé části aplikací(viz kapitola 4.4).



Obrázek 4: Proces porovnávání dvou projektů

Proces porovnávání se skládá dohromady ze čtyř základních kroků (viz obrázek 4). Každý krok procesu je povinný a je nemožné pokračovat v následujícím kroku dokud není současný krok dokončen.

1. Systém Maus může specifikovat vstupní hodnoty hned několika způsoby (viz kapitola 4.2), avšak z pohledu procesu porovnávání se bude vždy jednat o dvojici komprimovaných projektů.

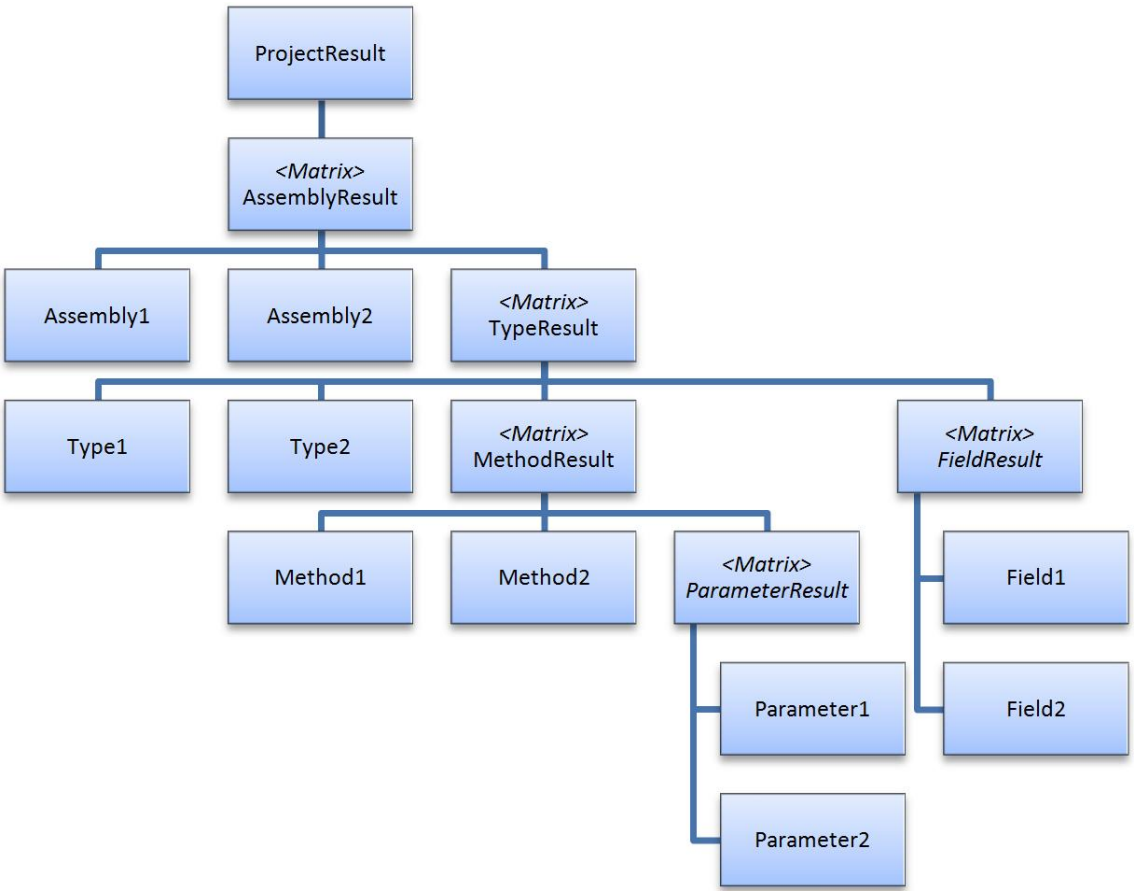
2. Následuje krok načtení všech assembly do paměti. Rozbalování komprimovaných projektů probíhá v paměti, bez nutnosti vytváření dočasných souborů na pevném disku. Při rozbalování se prochází archív a pro soubory odpovídajícího formátu (`.exe` nebo `.dll`) se vytvoří objektová reprezentace assembly. [1] Po dokončení de-

Obrázek 5: Načítání assembly

[1] Pokud vyrovnávací paměť obsahuje seznam assembly pro požadovaný projekt, kroky 2, 3 a 4 se přeskočí a vrátí se seznam z paměti.

komprimace se seznam získaných assembly zaznamená do paměti cache příslušné k identifikačnímu číslu projektu. To později umožní znovunačtení stejných assembly bez nutnosti načítání projektu z databáze a jeho rozbalování. Seznam získaných assembly je nakonec předán do následujícího kroku procesu.

3. Krok extrakce dílčích podobností má za cíl získat jednotlivé detaily z obou aktuálně zkoumaných assembly a také mezi těmito detaily vypočítat relativní podobnosti. O získávání detailů se stará třída `CSharpExtractor`, která implementuje rozhraní `IExtractor`. Za pomoci reflexe postupně vytváří *výsledkovou hierarchii objektů*, kterou inicializuje dostupnými informacemi o porovnávané dvojici. Tyto informace jsou uchovávány skupinou informačních tříd(viz kapitola 4.5). Pro správnou funkčnost je po inicializaci každého *výsledkového objektu* zapotřebí provést extrakci podobností. K tomu slouží metoda `ExtractSimilarities`, která je definovaná společným předkem každé *výsledkové třídy*.



Obrázek 6: Hierarchie výsledkových objektů

Pokud by se plánovalo rozšíření systému o podporu porovnávání aplikací na platformě Java, `IExtractor` by bylo rozhraní, které by pomyslná třída `JavaExtractor` musela implementovat.

4. Finální krok procesu porovnávání má za úkol nalézt právě takové dvě assembly/třídy/proměnné/metody/parametry, které mají spolu nejvíce společného. K tomuto účelu slouží právě dva algoritmy nad maticí *výsledkových objektů* (viz kapitoly 4.6.1 a 4.6.2). Vybrané páry jsou pak použity pro výpočet absolutní podobnosti projektů, které se dosahuje skládáním relativních podobností. Pro skládání relativních podobností se používá váženého průměru, kde jednotlivé váhy jsou předem načteny za pomocí managementu nastavení(viz kapitola 4.8).

4.4 Hierarchie výsledkových tříd

Tato skupina tříd se společným jmenným prostorem[1] umožňuje sestrojit hierarchii objektů, detailně popisující podobnosti mezi jednotlivými aspekty aplikací. Každá třída z tohoto jmenného prostoru má pro větší přehlednost zaveden sufix `Result` a jsou nadále v textu označovány za *výsledkové třídy/objekty*. Všechny výsledkové třídy jsou potomkem `BaseResult`[2]. Každá instance konkrétní výsledkové třídy zaobaluje právě dva prvky a popisuje detailní podobnosti mezi nimi.

Třída `BaseResult` definuje všechny základní operace výsledkových tříd a celočíselnou vlastnost `Result`, představující procentuální shodu dvou porovnávaných prvků. Dále obsahuje pomocnou vlastnost `Results`, která slouží pro skladování relativních výsledků v páru s jejich váhami.

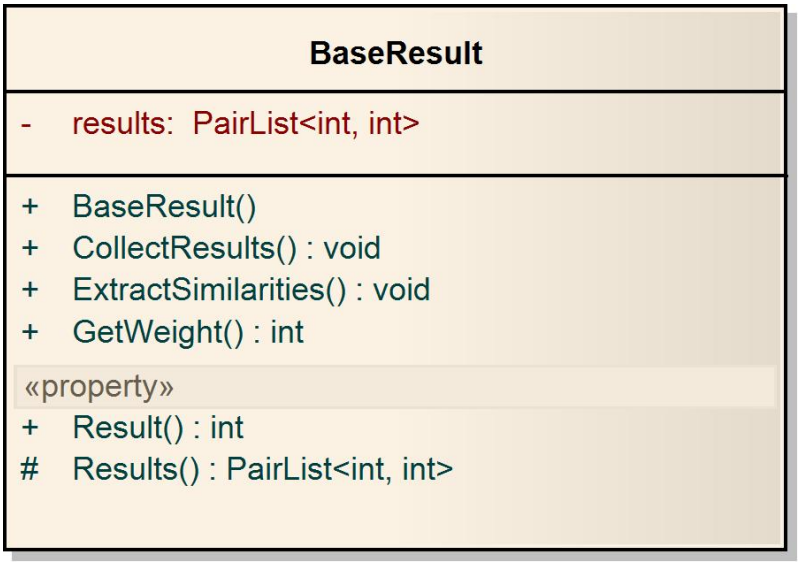
Pro interní účely projektu byla vytvořena generická třída `Pair`, která svým způsobem odpovídá struktuře `KeyValuePair` ze jmenného prostoru `System.Collections.Generic`. Tato třída je především interně využita pro implementaci generického listu `PairList`(viz obrázek 8).

`BaseResult` definuje tří základní operace, společné všem výsledkovým třídám, jako virtuální metody. Tyto metody jsou:

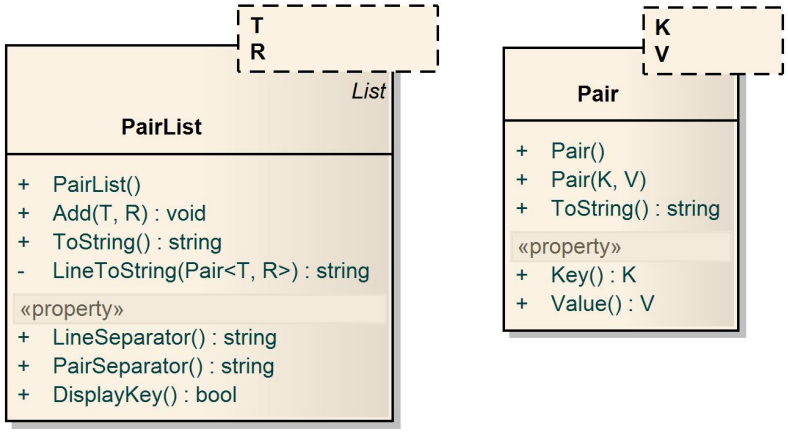
1. **CollectResults**: Slouží pro sběr dílčích výsledků porovnávání. Základní implementace v `BaseResult` obsahuje pouze výpočet váženého průměru z hodnot ucho-

[1] Veškeré výsledkové třídy se nacházejí ve jmenném prostoru SimilarityComparator.Results

[2] Jedinou vyjímkou je samotná třída BaseResult, která je také označována za výsledkovou třídu.



Obrázek 7: Společný předek - SimilarityComparator.Results.BaseResult



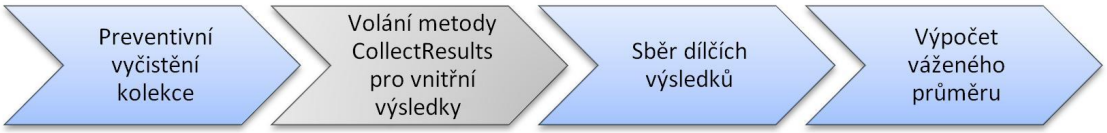
Obrázek 8: PairList - generický list párů.

vaných ve vlastnosti `Results` a uložení výsledku výpočtu do vlastnosti `Result`. Každý potomek třídy `BaseResult` by měl překrýt implementaci této metody a zajistit následující kroky:

- Vyčistit kolekci `Results`. To nám zaručí, že veškeré hodnoty v kolekci jsou opravdu aktuální, a že nepatří k předchozím výpočtům.
- Vložení dílčích podobností do kolekce `Results` s odpovídající váhou.
- Pokud dílčí podobnost není konečná(konkrétní číslo v rámci aktuální úrovně v hierarchii) a tudiž se jedná o *výsledkový objekt*, je nutné zavolat

`CollectResults` výsledkového objektu ještě před vložením výsledku do kolekce `Results`.

- Zavolat původní implementaci metody `CollectResults` z předka, což zajistí výpočet váženého průměru a jeho dosazení do vlastnosti `Result`.



Obrázek 9: Překrytí metody CollectResults

Z předchozího výčtu lze shrnout, že metoda `CollectResults` v podstatě nastaví hodnotu vlastnosti `Result` dle výsledku váženého průměru všech dílčích podobností. Výpis 4 ukazuje jednoduchou implementaci metody `CollectResults` v potomku třídy `BaseResult`.

```
public override void CollectResults()
{
  Results.Clear();

  TypeMatrix.CollectResults();
  Results.Add(TypeMatrix.Result, 1);

  base.CollectResults();
}
```

Výpis 4: Ukázka implementace metody CollectResults

2. **ExtractSimilarities**: Metoda slouží k extrakci podobností z dvojice prvků[3], které byly poskytnuty některou z implementací rozhraní `IExtractor`. Jednotlivé implementace `IExtractor` se starají především o poskytnutí konkrétních informací o aplikaci, jako jsou například název proměnné, její typ a podobně. Po poskytnutí těchto informací se navíc musí postarat o zavolání metody `ExtractSimilarities` pro každý *prvek* aplikace. Tím se porovnají jednotlivé detaily a výslednou procentuální shodu vloží do předem určené vlastnosti výsledkové třídy.

[3]Prvkem je myšlena jakákoliv část aplikace k porovnání. Tou mohou být následující části: assembly, třídy, metody, parametr metody, proměnné atd.

Obrázek 10: Použití metody ExtractSimilarities

Jelikož je extraktor `CSharpExtractor` pevně součástí vyvíjeného modulu, byl u všech potřebných výsledkových tříd implementován interní konstruktor, který se stará o inicializaci objektu a současně o zavolání metody ExtractSimilarities hned poté co inicializace skončí. Pokud by se plánovalo rozšíření mimo hranice tohoto modulu, bylo by nutné provést jak inicializaci, tak zavolání metody `ExtractSimilarities` manuálně.

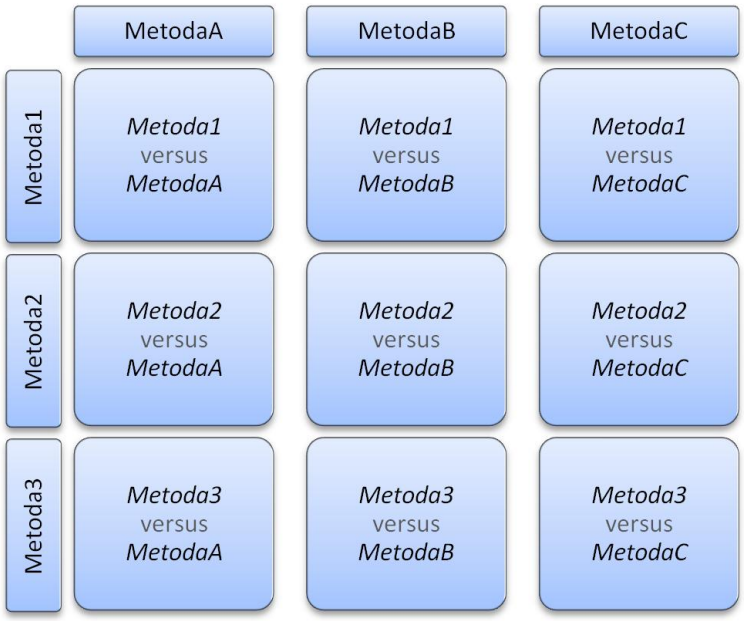
3. **GetWeight**: Tato metoda byla zavedena pro podporu rozpoznání vyjímek v prvcích. Vrací celočíselnou hodnotu, reprezentující váhu daného prvku. Pokud metoda vrátí nulovou hodnotu, bude tento prvek při dalších výpočtech ignorován. V současné době se této možnosti využívá pouze u tříd a metod, díky čemuž se dá konfigurací určit, které z těchto prvků nemají mít vliv na konečný výsledek porovnávání (viz kapitola 4.8). Využití se najde především pro generované části kódů, které by mohli negativně ovlivnit relevantnost výsledku porovnávání.

Hierarchie výsledkových objektů se během výpočtů mění. Inicializací se vytvoří složitá struktura, složená z několika vnořených matic. Každá matice představuje kombinaci prvků mezi právě dvěma o úroveň vyššími prvky.

Příklad 4.1
Mějme třídy Třída1 a TřídaA. Třída1 obsahuje metody Metoda1, Metoda2 a Metoda3. TřídaA obsahuje metody MetodaA, MetodaB a MetodaC. Matice vytvořená pro metody těchto dvou tříd je znázorněna na obrázku 11. ■

V průběhu porovnávání jsou jednotlivé kombinace v maticích eliminovány a konečným výsledkem je pouhý seznam párovaných prvků. Proces redukce kombinací z množiny popsané maticí na seznam relevantních kombinací je zajištěn algoritmy popsanými v kapitolách 4.6.2 a 4.6.1.

Návrh výsledkových tříd je odrazem požadavků, které jsou částečně vystihnuty obrázkem 6, ten popisuje způsob, jakým se má hierarchie objektů skládat dohromady.



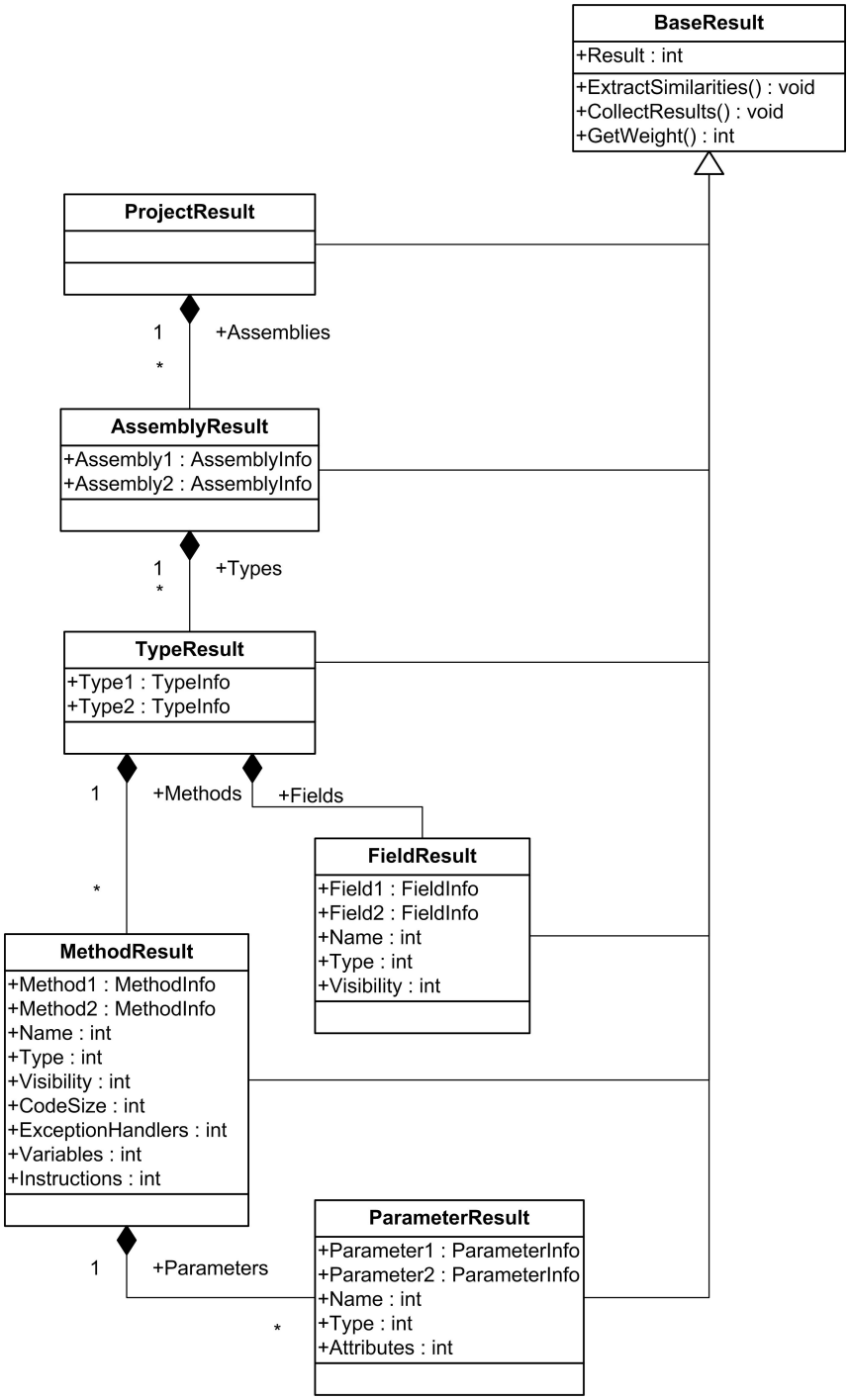
Obrázek 11: Matice metod k příkladu 4.1

Pro větší přehlednost si prvně představíme formu výsledkových tříd, potřebnou na konečném výstupu všech výpočtů. Tento základní návrh je zachycen třídním diagramem na obrázku 12.

Na tomto obrázku je názorně vidět, že díky společnému předku `BaseResult` sebou nesou všechny výsledkové třídy informaci o podobnosti prostřednictvím vlastnosti `Result`. Kořenovou třídou je `ProjektResult`. Ta obsahuje seznam všech podobných assembly. Každý podobný pár assembly reprezentovaný třídou `AssemblyResult` pak obsahuje seznam podobných typů[4] v rámci právě těchto dvou assembly. `TypeResult` dále obsahuje dva oddělené seznamy podobných prvků. Jeden seznam obsahuje informace o podobných metodách a druhý o proměnných, viditelných na úrovni daného typu. Ke každé dvojici podobných metod je navíc vytvořen seznam podobných parametrů. Informace o podobnostech mezi dvěma parametry nese třída `ParameterResult`.

Všechny tyto seznamy v konečném důsledku tvoří výslednou podobnost mezi dvojící projektů. Na začátku procesu porovnávání jsou však všechny tyto seznamy prázdné. O jejich naplnění se starají algoritmy, které zpracovávájí informace vložené do matic při ini-

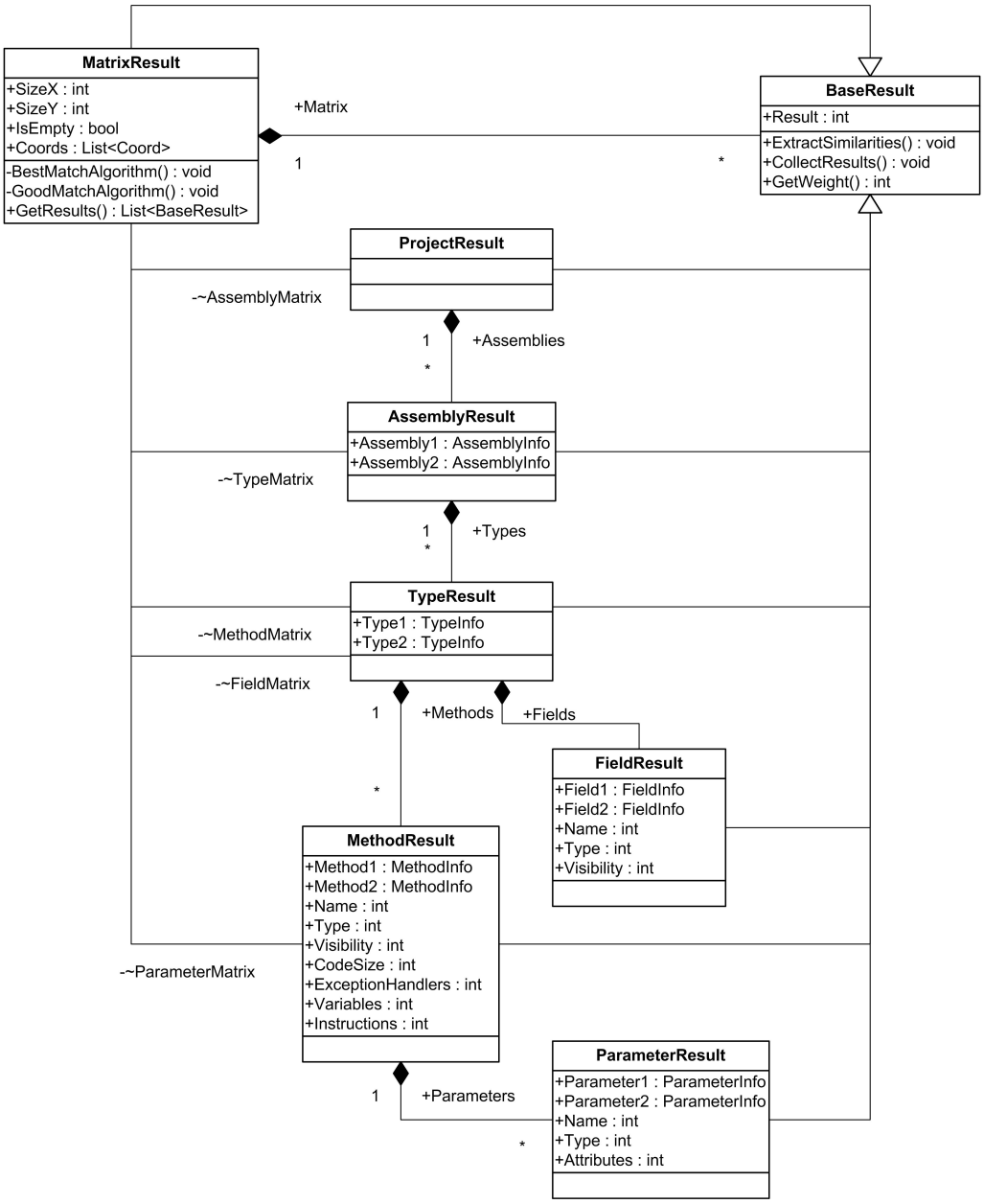
[4]Typem jsou myšleny jak referenční(class), tak i hodnotové(struct) typy.



Obrázek 12: Třídní diagram - Zjednodušená hierarchie výsledků

cializaci. Tyto matice jsou, stejně jako seznamy, přítomné na každé úrovni této hierarchie

tříd. Kompletní diagram hierarchie výsledků, obsahující matici jako kompozitní třídu, lze vidět na obrázku 13.

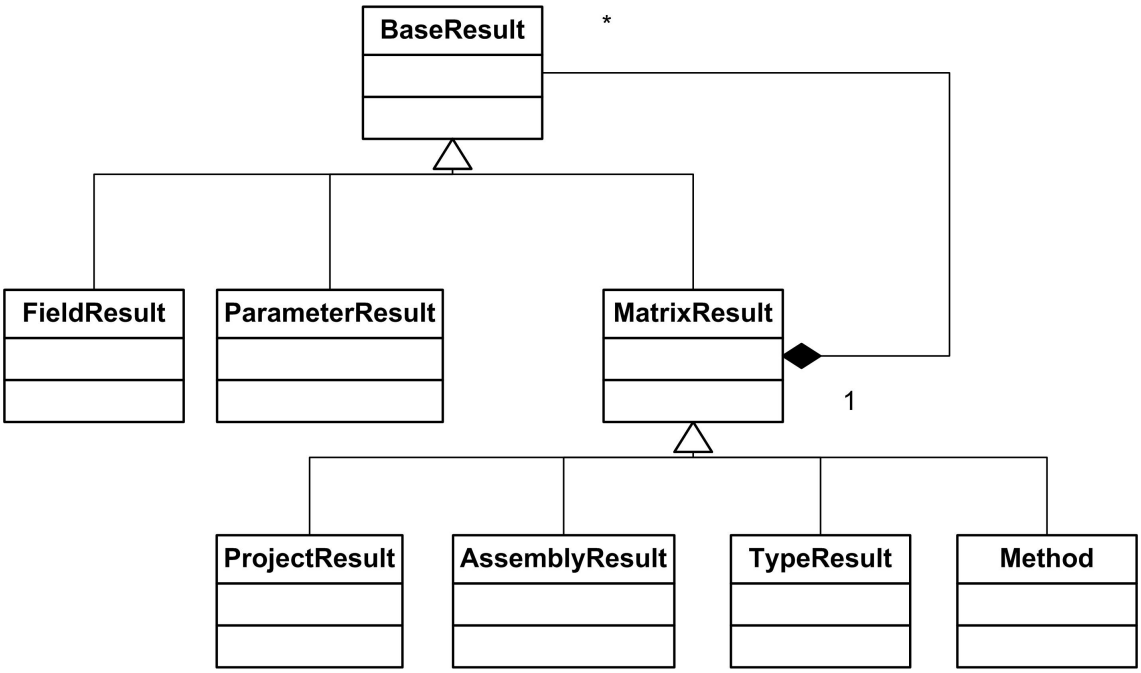


Obrázek 13: Třídní diagram - Kompletní hierarchie výsledků

Jedná se alternativní aplikaci návrhového vzoru kompozit. Matice je speciálním případem kompozitu, který skládá dohromady vždy stejné výsledkové třídy. Třídy jako

jsou například TypeResult, nebo MethodResult pak obsahují tyto homogenní matice. Proto by jsme neprávem označovali ostatní výsledkové třídy za listy návrhového vzoru kompozit.

Původní návrh(viz obrázek 14), který mnohem více zohledňoval návrhový vzor kompozit, měl hlavní nedostatek na úrovni třídy "TypeResult", která má obsahovat jak informace o proměnných tak i o metodách. Původní myšlenku dědění z MatrixResult třídy proto nahradil návrh s obecnou maticí, která je poté asociována s konkrétní výsledkovou třídou, jak ukazuje diagram na obrázku 13.



Obrázek 14: Třídní diagram - Původní myšlenka hierarchie výsledků

4.5 Informační třídy

Informační třídy slouží k uchování konkrétních informací o daném prvku. Všechny tyto třídy se nacházejí ve jmenném prostoru:

`SimilarityComparator.Results.InfoClasses`.

Třída	Vlastnost : Typ	Pozn.
BaseInfo	Name : string	Jméno prvku. Může se jednat např. o název metody, třídy, proměnné atd.
	Signature : string	Informace popisující prvek jako celek. Většinou se jedná o složení několika vlastností prvku dohromady. Např.: návratová hodnota + název metody + parametry.
AssemblyInfo	FullName : string	Celý název assembly, obsahující mimo jména také např. verzi assembly, nebo kulturu.
	Runtime : string	Runtime, pro který je assembly určena. Např. NET_2_0
	Kind : string	Druh assembly, který může nabývat hodnot: Dll, Console, nebo Windows
TypeInfo	FullName : string	Celý název typu je složen z jeho jmeného prostoru a názvu typu odděleného tečkou.
	Namespace : string	Jmenný prostor typu.
MemberInfo[a]	ReturnType : TypeInfo	návratový typ metody, popřípadě typ vlastnosti/-proměnné.
	DeclaringType : TypeInfo	Typ ve kterém je člen deklarovaný.
	Attributes : long	Celočíselná hodnota, vypovídající o atributech přeřazených třídnímu členu.
	IsStatic : bool	Indikuje, zdali se jedná o statického člena třídy.
	IsPublic : bool	Indikuje, zdali se jedná o veřejného člena třídy.
FieldInfo	HasDefault : bool	Indikuje, zdali má proměnná explicitně určenou počáteční hodnotu.
MethodInfo	CodeSize : long	Velikost kódu metody.
	ExceptionHandlers : int	Počet obsluhovaných vyjímek v těle metody.
	Variables : int	Počet proměnných deklarovaných v těle metody.
	HasBody : bool	Indikuje, zdali má metoda tělo.
ParameterInfo	Type : TypeInfo	Typ parametru.
	Attributes : long	Celočíselná hodnota, vypovídající o atributech přeřazených parametru.

Tabulka 2: Podporované informační třídy

[a] Obecný třídní člen. Např. metoda, vlastnost/proměnná.

Stejně jako tomu bylo u výsledkových tříd, mají všechny informační třídy společného předka. Tím je třída `BaseInfo`, obsahující obecné informace, společné všem zkou-

maným prvkům. Tabulka 2 obsahuje všechny podporované prvky, k nim relevantní informační třídy a seznamy vlastností prvků, jenž jsou tyto třídy schopny uchovávat.

4.6 Maticové algoritmy

Jak by se dalo usoudit z kapitoly 4.4, matice reprezentovaná třídou `MatrixResult` je nejdůležitějším prvkem celé hierarchie výsledků. Nejsložitějším krokem v celém procesu porovnávání by se dal označit úkol nalezení takové podmnožiny kombinací z matice, která bude nejlépe popisovat shodnosti mezi dvěma prvky vyšší úrovně.

Jako první by zřejmě každého napadlo použít všechny kombinace a pak předpokládat, že výsledkem bude alespoň trochu relevantní hodnota. Implementace takového algoritmu by byla nejjednodušší a jeho rychlost by byla více než uspokojující. Pravdou však je, že takový postup by znehodnotil celý proces porovnávání a to už jen z jednoho konkrétního důvodu - viz příklad 4.2.

Příklad 4.2
Mějme matici naplněnou hodnotami, které představující procentuální podobnost mezi metodami M_1, M_2, M_3 a M_A, M_B, M_C(viz obrázek 15).

	M_1	M_2	M_3
M_A	10%	30%	100%
M_B	50%	100%	10%
M_C	100%	40%	20%

Obrázek 15: Matice procentuálních podobností.

Jak si můžeme všimnout, metoda M_A je shodná s metodou M_3, M_B je podobná s M_2 a M_C s M_1. Proto bychom mohli říct, že třída obsahující metody M_1, M_2, M_3 je na 100% shodná s třídou obsahující metody M_A, M_B, M_C.

Pokud však vezmeme v úvahu všechny kombinace, jednoduchým aritmetickým průměrem všech v matici uvedených hodnot dostaneme shodnost 51%, což podle předchozí úvahy zřejmě není pravda. ■

Úkolem algoritmu tedy bude nalezení hodnot, které nejvíce vypovídají o podobnosti dvou prvků. Požadavky na algoritmus by se daly shrnout do dvou bodů:

1. Každá hodnota v matici vypovídá o podobnosti mezi dvojicí prvků. Pokud bude tato hodnota vybrána algoritmem, nemůže být vybrána již žádná hodnota související právě s těmito dvěma prvky.

2. Součet všech vybraných hodnot musí být nejvyšší možný.

V průběhu vývoje byly vytvořeny dva algoritmy řešící problém uvedený výše. Tyto algoritmy jsem pojmenoval podle jejich charakteristické funkce a jsou popsány v následujících dvou kapitolách.

4.6.1 Algoritmus dobré shody

Algoritmus dobré shody byl navržen v počátku vývoje modulu a požadavky na něj jsou následující:

- Jednoduchost implementace
- Rychlost
- Musí splňovat 1. požadavek na maticový algoritmus
- 2. požadavek na algoritmus může být splněný pouze částečně

Jeho princip zachycuje příklad 4.3.

Příklad 4.3
Mějme matici naplněnou stejnými hodnotami jako v příkladě 4.2. Pokud budeme vybírat hodnoty tak, aby splňovali výše zmíněné požadavky, budeme postupovat následovně:

1. Nalezneme nejvyšší možnou hodnotu. V tomto případě jsou to hned tři pozice s hodnotou 100%. Vybereme první z ních a označíme ji červeně, viz obrázek 16, matice vlevo.

2. Opakujeme první krok, avšak v prvním kroku jsme již vybrali jednu hodnotu a s ohledem na první pravidlo nesmíme použít žádnou hodnotu spojenou s prvky M_A a M_3. Proto hodnoty spojené s těmito prvky vyloučíme z výběru(hodnoty označené křížkem), viz obrázek 16, matice uprostřed.

3. Opakujeme druhý krok, dokud má matice dostupné hodnoty.

	M_1	M_2	M_3
M_A	10%	30%	**100%**
M_B	50%	100%	10%
M_C	100%	40%	20%

	M_1	M_2	M_3
M_A	~~10%~~	~~30%~~	~~**100%**~~
M_B	50%	**100%**	~~10%~~
M_C	100%	40%	~~20%~~

	M_1	M_2	M_3
M_A	~~10%~~	~~30%~~	~~**100%**~~
M_B	~~50%~~	~~**100%**~~	~~10%~~
M_C	**100%**	~~40%~~	~~20%~~

Obrázek 16: Výběr hodnot matice

■

Takto definovaný algoritmus bude mít výhodu v jeho rychlosti, avšak pokud se trochu zamyslíme, uvědomíme si, že za jistých okolností může být porušen druhý požadavek na algoritmus, který byl uvedený dříve. Jednu z těchto situací zachycuje příklad 4.4.

Příklad 4.4
Mějme opět matici o rozměrech 3x3, nyní však s trochu rozličnými hodnotami - viz obrázek 17.

	M_1	M_2	M_3
M_A	**91%**	10%	90%
M_B	90%	30%	10%
M_C	30%	80%	20%

	M_1	M_2	M_3
M_A	~~**91%**~~	~~10%~~	~~90%~~
M_B	~~90%~~	30%	10%
M_C	~~30%~~	**80%**	20%

	M_1	M_2	M_3
M_A	~~**91%**~~	~~10%~~	~~90%~~
M_B	~~90%~~	~~30%~~	**10%**
M_C	~~30%~~	~~**80%**~~	~~20%~~

Obrázek 17: Problém algoritmu dobré shody

Pokud budeme postupovat podle kroků uvedených v příkladu 4.3, dosáhneme postupně tří hodnot: 91%, 80% a 10%. Součet těchto hodnot nám dá 181 a průměrná hodnota bude přibližně 60%. Pokud se však pořádně podíváme na hodnoty v matici, můžeme si všimnout, že jsme právě porušili 2. pravidlo z požadavků na algoritmus. Součet těchto hodnot není nejvyšší možný. Nejvyšší možné hodnoty můžeme dosáhnout pokud budeme postupovat dle obrázku 18.

	M_1	M_2	M_3
M_A	91%	10%	**90%**
M_B	90%	30%	10%
M_C	30%	80%	20%

	M_1	M_2	M_3
M_A	~~91%~~	~~10%~~	~~**90%**~~
M_B	**90%**	30%	~~10%~~
M_C	30%	80%	~~20%~~

	M_1	M_2	M_3
M_A	~~91%~~	~~10%~~	~~**90%**~~
M_B	~~**90%**~~	30%	~~10%~~
M_C	~~30%~~	**80%**	~~20%~~

Obrázek 18: Správný výběr hodnot z matice.

Tato kombinace hodnot nám dává součtovou hodnotu 260 a průměrná hodnota vychází na přibližných 87%. ■

Jak jsme si ukázali v příkladu 4.4, algoritmus dobré shody nesplňuje oba požadavky na algoritmus, avšak z důvodů jeho rychlosti nebylo jeho využití úplně vyloučeno, ale pouze omezeno na nejnutnější případy. Ze stejného důvodu byl algoritmus právě pojmenován *algoritmem "dobré" shody*.

Časová náročnost tohoto algoritmu je určena následovně:

$$f(n) = O(n^3)$$

kde n představuje velikost delší strany matice.

Přesný zápis algoritmu je uveden ve výpise 5.

```
private void GoodMatchAlgorithm()
{
    for (int i = 0; i < NumberOfMatches; i++)
    {
        Coord lastCoord = null;
        int lastMaxNumber = 0;
        for (int x = 0; x < SizeX; x++)
        {
            for (int y = 0; y < SizeY; y++)
            {
                if (! IsAvailable(Coords, new Coord(x, y))) continue;
                // Přeskoč prvek, pokud má nulovou váhu.
                if (Matrix[x][y].GetWeight() == 0) continue;
                if (lastMaxNumber < Matrix[x][y].Result)
```

```
                {
                    lastMaxNumber = Matrix[x][y].Result;
                    lastCoord = new Coord(x, y);
                }
                // Pokud narazíme na hodnotu 100%,
                // není třeba pokračovat v procházení matice.
                if (lastMaxNumber == 100) break;
            }
            if (lastMaxNumber == 100) break;
        }
        if (null != lastCoord)
        {
            Coords.Add(lastCoord);
            lastCoord = null;
        }
    }
}
```

Výpis 5: Algoritmus dobré shody

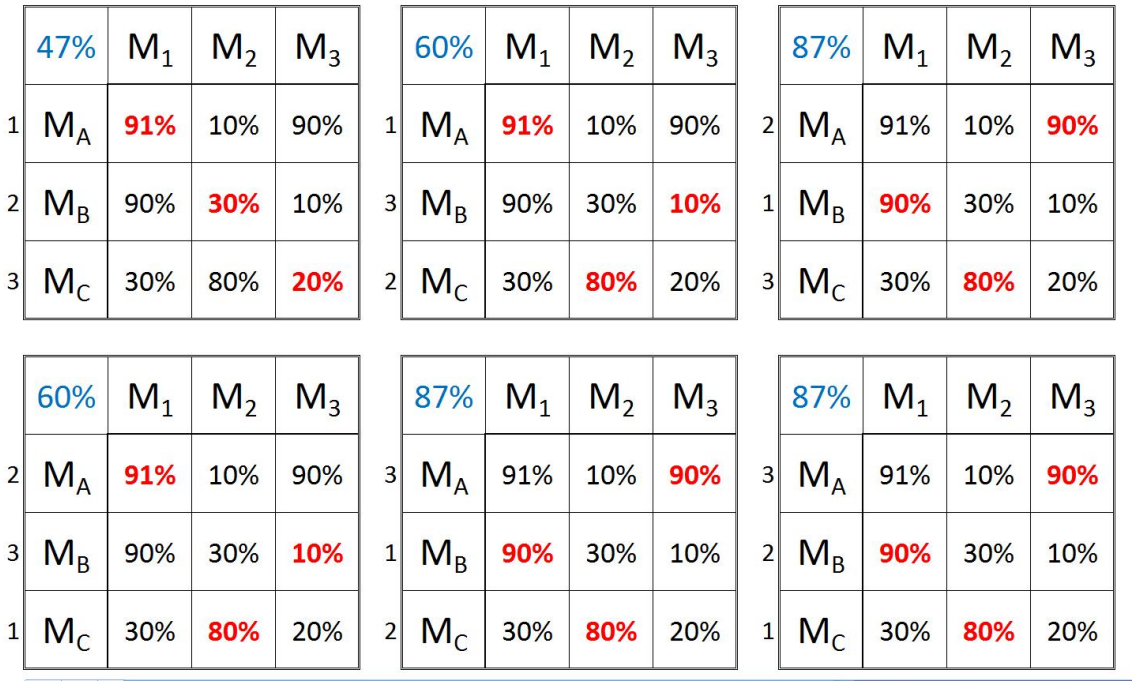
4.6.2 Algoritmus nejlepší shody

Algoritmus dobré shody popsaný v kapitole 4.6.1 by se dal charakterizovat jako velice rychlý, avšak ne vždy přesný algoritmus. Algoritmus nejlepší shody bude mít za úkol vyřešit právě problém nepřesnosti při vybírání nejlepší kombinace z matice.

Tento algoritmus je založený na zkoumání všech rozdílných výběrů, které vzniknou na základě rozdílného pořadí výběru řádků/sloupců v matici. Obrázek 19 znázorňuje 6 výběrů, které vzniknou permutací pořadních čísel řádků.
Algoritmus zkoumá řádek po řádku v pořadí, jaké určí vybraná permutace. V rámci řádku vybere nejlepší možnou hodnotu. Součástí výběru hodnoty v řádku, je také kontrola, zdali již nebyla v korespondujícím sloupci vybrána hodnota pro jiný řádek, tak aby se zajistilo splnění prvního požadavku na algoritmus.

Časová náročnost nalezení výběru pro jednu permutaci je téměř zanedbatelná. Mnohem větším problémem je otestování všech možných permutací. Jejich počet je dán vzorcem pro *permutaci bez opakování*:

$$P(n) = n!$$



Obrázek 19: Myšlenka algoritmu nejlepší shody

n	$P(n)$	Čas
1	1	< 0.01s
2	2	< 0.01s
3	6	< 0.01s
4	24	< 0.01s
5	120	< 0.01s
6	720	< 0.01s
7	5 040	< 0.01s
8	40 320	$\doteq 0.04$s
9	362 880	$\doteq 0.36$s
10	3 628 800	$\doteq 3.55$s
11	39 916 800	$\doteq 40$s
12	479 001 600	$\doteq 8$h

Tabulka 3: Permutace pro vybraná n

Jelikož faktoriál je velice rychle rostoucí funkce, bude využití tohoto algoritmu omezeno na předem určené maximální rozměry matice. Přehled permutací pro vybraná n s přibližnou časovou náročností shrnuje tabulka 3.

Z přehledu těchto hodnot lze stanovit maximální rozměr matice, pro kterou bude *algoritmus nejlepší shody* aplikovatelný. Optimální hranicí byla empiricky stanovena hodnota $n = 7$. To však neznamená, že algoritmus není aplikovatelný pro matice, které tuto hodnotu přesáhnou. Algoritmus nejlepší shody lze použít v případě, že alespoň jeden z rozměrů matice splňuje podmínku < 8. Algoritmus provádí permutace pouze v jedné dimenzi a tou dimenzí je právě ta s menšími rozměry. Pokud bude mít matice méně řádků jak sloupců, budou se permutace počítat nad množinou řádků. V opačném případě by byly pro tvorbu permutací zvoleny sloupce.

4.7 Podpůrné algoritmy

Maticové algoritmy 4.6.1 a 4.6.2 mají za úkol nalézt tu nejlepší možnou kombinaci prvků na základě jejich dílčích podobností. To znamená, že se ve stromové struktuře výsledkových objektů začíná od prvků, které ve stromu představují listy a postupným skládáním mezi-výsledků se dopracuje až ke kořenu stromu, jímž je objekt typu `ProjectResult`.

Výsledné procento podobnosti projektů je tedy ovlivňováno nejen maticovým algoritmem, ale také počátečními hodnotami, které představují listy stromu výsledků.

Tyto počáteční hodnoty jsou inicializovány metodou `ExtractSimilarities`, popsanou v kapitole 4.4. Ta má za úkol porovnat dva prvky, které jsou popsané pomocí dvou informačních objektů. Dílčími výsledky této metody jsou právě listové hodnoty stromu výsledků. K výpočtům se využívá podpůrných algoritmů, vyvinutých speciálně pro tento případ. Podůrné porovnávací algoritmy se nachází v knihovní třídě `GeneralComparator`, která poskytuje přetížené metody `Compare`. Vstupními parametry jsou převážně dva objekty stejného typu a výsledkem je celočíselná hodnota, představující procento shody mezi těmito dvěma porovnávanými objekty.

Podporované typy k porovnání jsou následující:

- `bool`: primitivní porovnání dvou booleovských hodnot. Návratová hodnota 100 pro dvě stejné hodnoty a 0 pokud jsou hodnoty rozdílné.
- `long/int`: porovnání dvou celočíselných hodnot. Použit stejný algoritmus jako pro `double`.
- `double/float`: porovnání dvou desetinných čísel(viz kapitola 4.7.1).

- `TypeInfo`: porovnání dvou typů(viz kapitola 4.7.2).
- `string`: porovnání dvou řetězců. Založeno na porovnávání dvou seznamů znaků algoritmem přibližné shody(viz `IList`).
- `IList`: Porovnání dvou seznamů objektů(viz kapitola 4.7.3).

4.7.1 Algoritmus číselné podobnosti

Značná část informací, které jsou na zdrojových kódech zkoumány, bývá popsána nějakou číselnou hodnotou. Pro podporu porovnávání informací tohoto druhu, byla vytvořena sada porovnávacích metod:

- `int Compare(long l1, long l2)`
- `int Compare(long l1, long l2, float border)`
- `int Compare(double d1, double d2)`
- `int Compare(double d1, double d2, float border)`

Všechny tyto metody jsou postaveny nad jedním algoritmem. Cílem algoritmu je získat procentuální podobnost dvou čísel, která se odráží na vzdálenosti těchto dvou čísel od sebe. Aplikování lineární závislosti se ukazuje jako nedostatečné řešení, proto je za klíčovou funkci vybrán *sinus*.

Příklad 4.5
Mějme dvě hodnoty $A = 32$ a $B = 38$, které budeme chtít porovnat.

Nyní volejme metodu `Compare`:

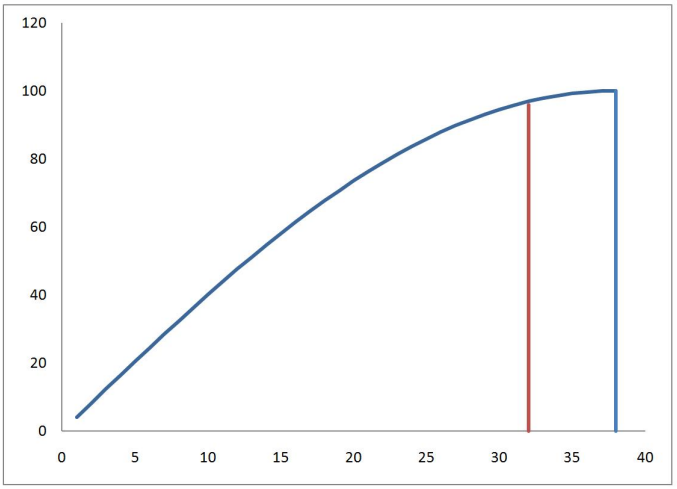
```
int výsledek = GeneralComparator.Compare(A, B);
```

Proměnná `výsledek` bude obsahovat hodnotu 97% (viz obrázek 20).

■

Příklad 4.6
Mějme opět dvě hodnoty $A = 32$ a $B = 38$, které budeme chtít porovnat. Dodatečně si deklarujeme hraniční bod $P = 0.6$.

Nyní volejme metodu `Compare`:

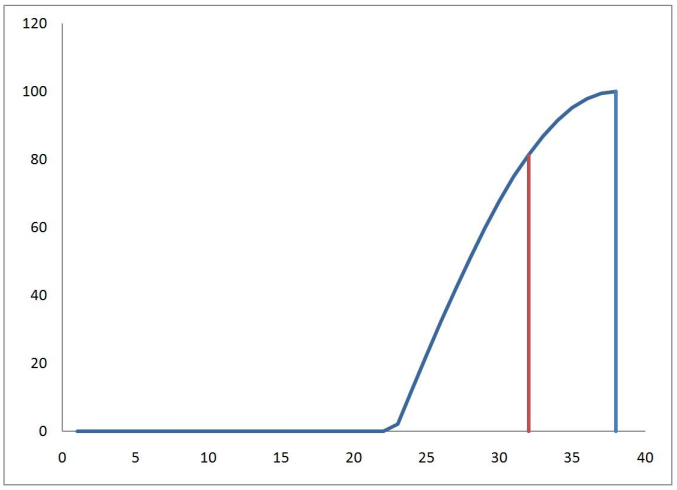


Obrázek 20: Porovnání čísel *algoritmem číselné podobnosti*

```
int výsledek = GeneralComparator.Compare(A, B, P);
```

Proměnná `výsledek` bude nyní obsahovat hodnotu 81% (viz obrázek 21).

■



Obrázek 21: Aplikace hraničního bodu při porovnání čísel algoritmem číselné podobnosti

Za pomoci hraničního bodu, lze detailně určovat citlivost pro jednotlivá porovnání. Hodnota hraničního bodu se pohybuje v rozmezí 0 až 1 a představuje hranici poměru dvou porovnávaných čísel, za kterou se tyto čísla berou jako rozdílná a tudíž je jejich podobnost nulová.

4.7.2 Podobnost typů

Způsob porovnávání typů jako "rovnost/nerovnost" se z praktických hledisek musí zavrhnout. Tvorba plagiátů totiž často vzniká záměnou typů za jejich ekvivalenty, popřípadě za typy, které mohou nést stejnou hodnotu, ale v jiné formě.

Princip porovnávání typů knihovní třídou `GeneralComparator` se založen na typových konverzích, tzv. přetypovávání. Úrovně možných přetypování a k ním přiřazené procentuální váhy zachycuje tabulka 4.

Úroveň	Název	Podobnost	Příklad
1	Ekvivalentní	100%	int a int
2	Obousměrný	80%	int a float
3	Jednosměrný	40%	int a string
4	Neekvivaletní	0%	int a ArrayList

Tabulka 4: Úrovně přetypování

4.7.3 Algoritmus přibližné shody

Algoritmus přibližné shody, dále jen AM algoritmus, je zobecněním algoritmu přibližného porovnání řetězců(ASM - Approximate String Matching)[11]. Ten je založen na počítání editační vzdálenosti, což je minimální počet elementárních úprav k tomu, aby oba řetězce byly ekvivalentní.

K výpočtu se používá matice celých čísel s rozměry určenými délkou porovnávaných řetězců. Na počátku je matice inicializována, jak popisuje tabulka 5.

		W	o	r	l	d
	0	1	2	3	4	5
W	1					
o	2					
r	3					
d	4					

Tabulka 5: ASM Algoritmus - počáteční matice

Matice M se následně řádek po řádku vyplňuje číselnými hodnotami podle následujícího pravidla:

$$M_{i,j} = \textbf{if } (x_i = y_j) \textbf{ then } M_{i-1,j-1} \textbf{ else } 1 + min(M_{i-1,j}, M_{i,j-1}, M_{i-1,j-1})$$

kde x a y jsou vstupní slova a funkce min vrací nejmenší z poskytnutých hodnot.

Aplikací tohoto pravidla na vstupní slova "World" a "Word" dostaneme matici uvedenou v tabulce 6.

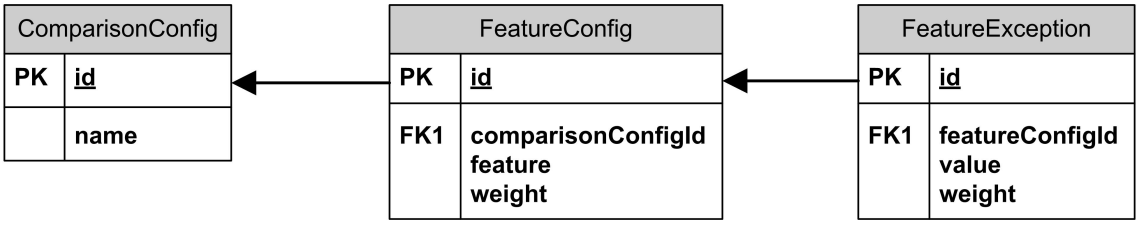
		W	o	r	l	d
	0	1	2	3	4	5
W	1	0	1	2	3	4
o	2	1	0	1	2	3
r	3	2	1	0	1	2
d	4	3	2	1	1	1

Tabulka 6: ASM Algoritmus - konečná matice

Hodnota $M_{5,4}$ určuje počet kroků nutných k získání ekvivalentních řetězců. V tomto případě je nutná pouze jedna úprava. Tato úprava může být reprezentována buď smazáním znaku 'l', nebo jeho vložením. Záleží na pohledu, z kterého řetězce je úprava brána.

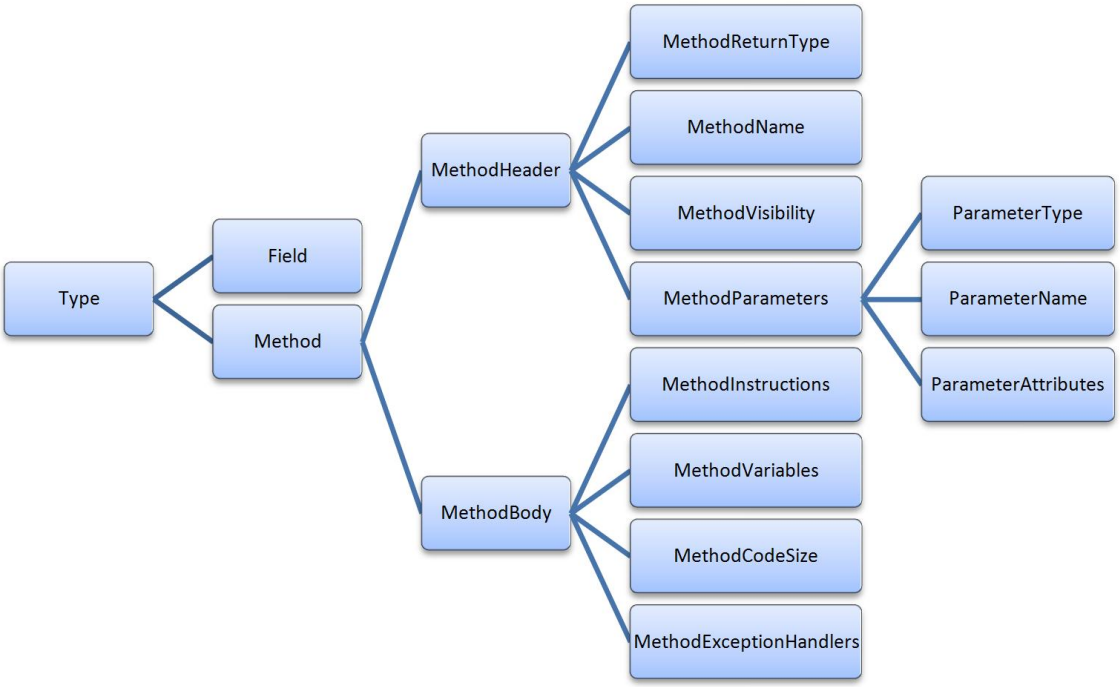
4.8 Konfigurace modulu

Některé studentské projekty jsou většího rozsahu, některé se naopak skládají např. jen z jedné jediné třídy. Některá zadání mohou předepisovat studentům implementaci určitého rozhraní, nebo naopak nechávat v oblasti implementace úplnou volnost. V každém případě se bude při hledání plagiátů zkoumat pokaždé něco trochu jiného. Na jisté rysy zdrojového kódu se bude dávat vyšší důraz a na některé zase nižší, nebo se v některých případech mohou jisté oblasti zcela ignorovat.
Aby bylo možné výpočet podobností upravovat dle situace, jsou k jednotlivým prvkům aplikace přiřazeny váhy, které umožňují určit, jak silný vliv bude mít prvek na celkové vyhodnocení podobnosti. Konfigurací těchto vah pak lze dosáhnout požadovaného chování. Základní návrh systému vah je zachycen na obrázku 22.



Obrázek 22: ER diagram - Systém konfigurace vah

Konfigurace je dána svým názvem. Systém pak může předat modulu informaci o tom, která z přednastavených konfigurací se má použít pro výpočet podobností. Každá konfigurace obsahuje seznam všech výpočtem podporovaných rysů aplikace a pro každý z těchto rysů definuje váhu. Tyto rysy jsou stejně jako prvky aplikace uspořádány do stromové struktury. Modul podporuje rysy aplikace uvedené na obrázku 23.



Obrázek 23: Hierachie podporovaných rysů

V rámci konfigurace každého rysu je možné zavést vyjímky, určující které konkrétní prvky mají mít rozdílnou váhu od všech ostatních prvků stejného druhu. Této funkcionality se především dá využít pro ignorování např. některých generovaných tříd, nebo metod.

5 Testování a nastavování modulu

Během vývoje byl modul průběžně testován na množině asi 40 studentských projektů. Díky testování se nalezlo mnoho nedostatků, které tak mohly být odstraněny. Na základě testování vznikly některé optimalizace algoritmů, které nejen umožnily zrychlení celého procesu porovnávání, ale také pozitivně ovlivnily přesnost výstupů.

Hlavním tématem této kapitoly však budou testy prováděné nad hotovým modulem za cílem vytvořit sadu konfigurací pro rozdílné druhy zadání studentských projektů.

n	**Počet projektů** n	**Počet porovnání** $C_2(n)$	**Název**
1	5	10	Anagramy
2	4	6	Analyzátor obrazu
3	4	6	Cesta v bludisti
4	1	0	Databáze CD
5	2	1	Dohled sítě
6	4	6	Elektronický diář
7	6	15	Grafický kalkulátor
8	1	0	Hanojské věže
9	2	1	Hodnost matice
10	2	1	Hra miny
11	4	6	Chat
12	1	0	Jednoduchý webový server
13	2	1	Komprese
14	4	6	Kontrolor souboru
15	4	6	Kreslítko
16	4	6	Kůň na šachovnici
17	2	1	Logika
18	3	3	Matematické cvičení
19	6	15	Maximální suma
20	5	10	Mince
21	5	10	Násobení čísel

Tabulka 7: Testované projekty - 1.část

Testovací množina obsahuje 135 studentských projektů. Projekty jsou před testováním rozdělovány do skupin dle otázky, kterou projekt řeší. Projekty s unikátním zadáním se tím pádem z procesu porovnávání vyhnou. Konečné množství testovaných projektů tedy je 129 a tyto jsou rozděleny do 36 skupin.

	Počet projektů n	**Počet porovnání** $C_2(n)$	**Název**
22	2	1	Nejkratší cesta v orientovaném grafu
23	2	1	Optimální násobení matic
24	1	0	Paintbrush
25	11	55	Palindromy s čísly
26	1	0	Pexeso
27	2	1	Piškvorky
28	3	3	Plocha
29	4	6	Porovnávání souborů
30	2	1	Pozice
31	2	1	Práce s tiskárnou
32	3	3	Převod na zlomek
33	3	3	Puzzle
34	4	6	Rozklad na prvočísla
35	2	1	Římská čísla
36	1	0	Síťové šachy
37	5	10	Šachovnice
38	4	6	Šachy
39	3	3	Tranzitivní uzávěr v grafu
40	3	3	Vyčisti zlomek
41	4	6	Vyhledávání souboru
42	2	1	Výrazy

Tabulka 8: Testované projekty - 2.část

Počet porovnání je v rámci každé skupiny dán počtem kombinací bez opakování.

$$C_k(n) = \binom{n}{k} = \frac{n!}{k!(n-k)!}$$

Pro proces porovnávání vždy platí $k = 2$, jelikož se porovnává po dvojicích. Výraz tudíž můžeme přepsat:

$$C_2(n) = \frac{n!}{2!(n-2)!} = \frac{n(n-1)(n-2)!}{2(n-2)!} = \frac{n(n-1)}{2}$$

5.1 Testovací prostředí

Pro testování časové náročnosti výpočtů byla použita následující sestava:

Operační systém	Windows Vista
Procesor	Intel® Core™ 2 Duo E7300 2.67GHz
Paměť(RAM)	4GB
Web Server	IIS 7.0
DB Server	MS SQL Server Enterprise Edition, win456.vsb.cz

Tabulka 9: Testovací prostředí

5.2 Časová náročnost

Porovnávání 135 poskytnutých projektů by nemělo trvat příliš dlouho a to hlavně z důvodu relativně malých skupin, do kterých byly rozděleny. Celkový počet 221 porovnávání vyplývá z hodnot uvedených v tabulkách 7 a 8.

Příklad 5.1
Pro ilustraci si představme, že projekty nebudeme rozdělovat do skupin, ale budeme je porovnávat všechny dohromady.

Stanovme si, že porovnání jedné dvojice bude trvat 1 vteřinu. Tudíž pro 221 původních porovnání by to znamenalo, že proces porovnávání bude celkově trvat 3 minuty a 41 vteřin. Nyní si vypočítejme, kolik porovnání by bylo nutné provést bez zavedení skupin projektů:

$$C_2(n) = \frac{n(n-1)}{2}$$

Když $n = 135$, pak:

$$C_2(135) = \frac{135 \cdot 134}{2} = 9045$$

To znamená, že porovnání takového množství projektů navzájem by trvalo 2 hodiny, 30 minut a 45 vteřin.

■

Reálné porovnávání těchto 135 projektů na sestavě uvedené v 5.1, probíhalo přibližně 2 minuty a 20 vteřin.

5.3 Testování přesnosti

Testování přesnosti probíhalo nad množinou všech v systému dostupných projektů. Postup testování, který jsem použil, se skládal ze dvou fází. V první fázi se spustilo porovnávání všech projektů a po dokončení výpočtů byly z databáze načteny výsledky. V druhé fázi jsem procházel výsledky a kontroloval, zda hodnoty získané k dané dvojici odpovídají faktům o zdrojových kódech, které jsem manuálně porovnal. Na základě této manuální kontroly byly nalezeny nedostatky původní konfigurace a ze znalostí těchto nedostatků jsem navrhl několik konfigurací, pro nejčastěji objevující se druhy projektů. Tyto konfigurace jsou součástí uživatelské příručky, která je k dispozici na přiloženém CD.

Testováním jsem potvrdil několik pozitivních nálezů plagiátorství. Kromě níže zmíněné dvojici projektů(viz kapitola 5.3.1) byly dále nalezeny tři dvojice téměř identických projektů. S výsledkem 93% byl potvrzen plagiát ve skupině "Anagramy", "Chat" a s podobností 92% ve skupině "Mince".

Falešný nález s podobností 87% byl potvrzen ve skupině "Převod na zlomek". Tak vysokému procentu podobnosti napomohla především obrovská převaha *vlastností* a *proměnných* nad samotnými metodami. Jelikož se v procesu porovnávání bere vlastnost jako metoda s prefixem "get_" popřípadě "set_", byla celková hodnota negativně ovlivněna podobností všech vlastností třídy.

Na porovnávání se také negativně projevilo velké množství metod pro obsluhu událostí u některých Windows Forms aplikací. Ty jsou totiž plnohodnotnými metodami a nelze je jednoduše odlišit od metod, které nesou řídíci logiku aplikace. Tento problém se projevil např. ve skupině "Grafický kalkulátor", který obsahuje spoustu grafických prvků a jen malá část programu řeší samotnou funkci aplikace.

V podobnostním rozsahu 65-80% se objevilo několik projektů ze skupiny "Palindromy s čísly", které měly podobně strukturovaný zdrojový kód, avšak samotné řešení se dostatečně lišilo k tomu, abych dokázal usoudit, že se nejedná o plagiáty.

U projektů s výsledkem porovnání okolo 50% a níže se ve všech případech jednalo o neslučitelné dvojice. Možnost plagiátorství u těchto projektů byla vyloučena.

5.3.1 Testování vybraných projektů

Pro ilustraci jsem vybral 4 projekty ze skupiny "Palindromy s čísly". Výběr těchto ukázkových projektů byl uzpůsoben tak, aby pokryl co nejvyšší rozsah v měřítku podobností. Níže uvedené projekty jsou součástí přiloženého CD.

Pro další výklad budou projekty označovány pořadovým číslem. Zde je shrnuta jejich stručná charakteristika:
Projekt 1:

- jedna třída
- pět statických metod
- metoda main obsahuje velkou část řídící logiky

Projekt 2:

- v podstatě se jedná o kopii projektu 1
- provedeny drobné úravy - převážně na úrovni 2 (viz tabulka 1)

Projekt 3:

- jedna třída
- čtyři statické metody
- signatury metod podobné s projekty 1 a 2
- rozdílný přístup k implementaci řešení oproti projektům 1 a 2

Projekt 4:

- dvě třídy (metoda main oddělená od řešení)
- třída s řešením obsahuje šest metod (nestatických) a dvě globální proměnné
- signatury metod odlišné od ostatních řešení
- rozdílný přístup k implementaci řešení oproti projektům 1, 2 a 3

Cílem testování je určit hodnoty konfigurace tak, aby výsledek porovnávání vypovídal o následujících faktech:

- Projekty 1 a 2 jsou téměř identické
- Projekt 3 je podezřele podobný s projekty 1 a 2
- Projekt 4 je odlišný od projektů 1, 2 a 3

Aplikací výchozího nastavení vah pro porovnávání lze získat podobnosti, které popisuje tabulka 10.

Porovnávané projekty	**Podobnost(%)**
1 a 2	98
1 a 3	79
2 a 3	81
1 a 4	55
2 a 4	56
3 a 4	59

Tabulka 10: Výsledky porovnávání pro výchozí nastavení

Na základě zkoumání dílčích podobností byly identifikovány nedostatky výchozí konfigurace a vytvořena konfigurace nová. Tato konfigurace se od výchozího nastavení liší v následujících bodech:

- Jména identifikátorů mají nulovou váhu.
- Metoda má mnohonásobně vyšší váhu než globální proměnná.
- Nejvyšší váhu v hlavičce metody mají parametry a typ návratové hodnoty.
- U parametru metody má nejvyšší váhu jeho typ. Jméno parametru a jeho atributy jsou podřadné.
- V těle metody mají největší váhu instrukce.
- Metoda `Main` je při porovnávání ignorována.

Výsledky porovnávání pro nově vytvořenou konfiguraci jsou shrnuty v tabulce 11.

Porovnávané projekty	Podobnost(%)
1 a 2	100
1 a 3	72
2 a 3	72
1 a 4	44
2 a 4	44
3 a 4	47

Tabulka 11: Výsledky porovnávání pro alternativní konfiguraci

5.4 Rozbor výsledků

Vedlejším efektem metody porovnávání aplikací na úrovni *mezikódu* je, že při porovnávání detailů kódu se často najdou podobné rysy a to i u kódů, které jsou výrazně odlišné. To má za následek zvýšení celkové podobnosti dvou porovnávaných projektů. Proto se jen zřídka najde dvojice projektů, která by měla podobnost nižší než 40%. Pro odstranění tohoto nedostatku řešení by bylo zapotřebí dlouhodobých testů a ladění podpůrných algoritmů popsaných v kapitole 4.7.

Na základě rozsáhlých testů, analyzování zdrojových kódů a s nimi souvisejících výsledků, byly empiricky stanoveny úrovně podobností. Tyto úrovně s vysvětlením jejich významu jsou uvedeny v tabulce 12.

Podobnost(%)	Význam
100-91	*Téměř identické*. U takové dvojice projektů je více než pravděpodobné, že se jedná o plagiát.
90-81	*Velmi podezřelé*. Pravděpodobně se jedná o plagiát.
80-66	*Podezřelé*. Může se jednat např. o částečně opsané řešení, nebo o řešení velmi podobné.
65-50	*Rozdílné řešení*. Některé pasáže mohou být podobné, avšak s největší pravěpodobností se jedná o náhodu.
50-0	*Odlišné*. Jedná se o dvojici projektů, které spolu mají společné pouze jen velice málo. S vysokou pravděpodobností se nejedná o plagiát.

Tabulka 12: Úrovně podobností

Pro projekty s podobností vyšší než 65% platí, že by měly být podrobeny detailnějšímu zkoumání pedagogem.

6 Závěr

Předmětem této diplomové práce bylo navrhnout a implementovat modul pro kontrolu jedinečnosti studentských projektů. V úvodu jsem analyzoval problémy řešené při porovnávání aplikací a na základě získaných znalostí jsem navrhl a implementovat požadovaný modul. Vývoj modulu probíhal v krátkých iteracích, tak abych byl schopný se přizpůsobit změnám, které vznikaly na základě zkušeností z implementační fáze. Některé části, jako jsou např. struktura výsledkových tříd, jádro modulu i jeho rozhraní, byly několikrát změněny tak, aby vyhovovaly požadavkům systému Maus. Tato práce popisuje návrh z poslední známé iterace vývoje modulu.

V rámci testování byla ukázána využitelnost modulu pro praktické účely, ale také byly odhaleny některé jeho nedostatky a oblasti, ve kterých modul vyžaduje dodatečná vylepšení. Testováním byly dále identifikovány konfigurace odpovídající nejčastějším druhům projektů pro vybrané předměty. Pro potřeby konfigurace modulu byla také napsána uživatelská příručka, která je obsažena na přiloženém CD.

7 Reference

[1] Lutz Prechelt, Guido Malpohl, Michael Phlippsen, *JPlag: Finding plagiarisms among a set of programs*, Fakultät für Informatik, Universität Karlsruhe, Germany, 2000.

[2] Maurice Howard Halstead, *Elements of Software Science*, Operating and Programming Systems Series. Elsevier North-Holland, New York, 1977.

[3] Jeong-Hoon Ji, Gyun Woo, Hwan-Gue Cho, *A source code linearization technique for detecting plagiarized programs*, New York: ACM, 2007.

[4] Karl J. Ottenstein, *An algorithmic approach to the detection and prevention of plagiarism*, New York: ACM, 1976.

[5] Christian Arwin, S. M. M. Tahaghoghi, *Plagiarism detection across programming languages*, Australia: Australian Computer Society, Inc., 2006.

[6] David Gitchell, Nicholas Tran, *Sim: a Utility For Detecting Similarity in Computer Programs*, New York: ACM, 1999.

[7] Michael J. Wise, *YAP3: improved detection of similarities in computer program and other texts*, New York: ACM, 1996.

[8] *Standard ECMA-335 - Common Language Infrastructure (CLI) 4th edition*, ECMA, 2006
`http://www.ecma-international.org/publications/standards/Ecma-335.htm`

[9] *Common Intermediate Language*, Wikipedia
`http://en.wikipedia.org/wiki/Common_Intermediate_Language`

[10] *Whats the difference between MSIL and CIL?*, .NET Uncle
`http://www.dotnetuncle.com/Difference/32_msil_cil_type.aspx`

[11] Gonzalo Navarro, *A Guided Tour to Approximate String Matching*, New York: ACM, 2001.

[12] Marco Gori , Marco Maggini , Lorenzo Sarti, *Exact and Approximate Graph Matching Using Random Walks*, IEEE, 2005.

[13] Vic Ciesielski, Nelson Wu, Seyed Tahaghoghi, *Evolving Similarity Functions for Code Plagiarism Detection*, Atlanta: ACM, 2008.

[14] Martin Dick, Judy Sheard, Cathy Bareiss, Janet Carter, Donald Joyce, Trevor Harding, Cary Laxer, *Addressing Student Cheating: Definitions and Solutions*, New York: ACM, 2002.

[15] Steve Engels, Vivek Lakshmanan, Michelle Craig, *Plagiarism Detection Using Feature-Based Neural Networks*, New York: ACM, 2007.

[16] Thomas Lancaster, Fintan Culwin, *Towards an Error Free Plagiarism Detection Process*, New York: ACM, 2001.

[17] Chao Liu, Chen Chen, Jiawei Han, Philip S. Yu, *GPLAG: Detection of Software Plagiarism by Program Dependence Graph Analysis*, New York: ACM, 2006.